UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS FÍSICAS
Departamento de Arquitectura de Computadores y Automática

TÉCNICAS DE UBICACIÓN DE TAREAS Y DEFRAGMENTACIÓN PARA MULTIÁREA HARDWARE EN SISTEMAS DINÁMICAMENTE RECONFIGURABLES.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Jesús Tabero Godino

Bajo la dirección de los doctores

Julio Septién del Castillo
Hortensia Mecha López

Madrid, 2010

- **ISBN: 978-84-693-1842-3**

Técnicas de ubicación de tareas y defragmentación para multitarea hardware en sistemas dinámicamente reconfigurables

---

Jesús Tabero Godino

Tesis Doctoral

Universidad Complutense de Madrid

Dpto. Arquitectura de Computadores y Automática

Técnicas de ubicación de tareas y defragmentación para multitarea hardware en sistemas dinámicamente reconfigurables

Memoria presentada por Jesús Tabero Godino para optar al grado de doctor por la Universidad Complutense de Madrid, realizada bajo la dirección de D. Julio Septién del Castillo y Dña. Hortensia Mecha López.

Madrid, Abril 2009

Este trabajo ha sido posible gracias a la financiación de la Comisión Interministerial de Ciencia y Tecnología, a través del proyecto CICYT TIC 2002-00160, TEC2005-04752 y TW2006-03274.

Agradecimientos

Han sido casi siete años en los que mucha gente me ha ayudado y me ha influido tanto a nivel personal como profesional.

Gracias, primero que todo a mi mujer Paloma, a ella especialmente le dedico esta Tesis porque siempre ha estado apoyándome, moral y físicamente, en especial dándole la atención a mis hijos que en muchos momentos no les he podido prestar. A mis hijos, Lydia, Pablo y Diego, para que sepan que hay que crecerse ante la adversidad (en muchas ocasiones me tuve que emplear a fondo para aislarme del ruido mientras jugaban).

También a mis queridos padres, que no tienen idea de qué diablos estudié (lo digo con mucho cariño), y que sólo con decirles el título de la Tesis les asusto, pero que si no fuera por el apoyo moral y lo más importante, por no presionarme a estudiar otra cosa, por respetar mis decisiones y confiar siempre en mí, cuando he abordado cualquier proyecto. Gracias también a mis hermanos porque siempre nos hemos apoyado mutuamente, y por estar ahí siempre, para todo.

A mis compañeros del Instituto por disculparme cuando estaba ausente, y en especial a mi compañero de despacho Jesús, por contrastar, opinar y escucharme.

Por último y no menos importante, agradecer a Julio, Hortensia y Daniel toda su paciencia y consejos que me han ido dando a lo largo de este tiempo. Por sus correcciones exhaustivas, por los acertados análisis técnicos sobre mis propuestas y ensayos o la aportación de su

experiencia en las publicaciones y en la tesis. Ellos han sabido darme ánimos en los momentos más difíciles, y siempre me han dedicado su atención y tiempo.

Capítulo 1: Introducción

El objetivo de este trabajo de investigación es la gestión de **recursos hardware dinámicamente reconfigurables (HWDR)** de gran capacidad para permitir multitarea Hardware **(HW)**. Dicha multitarea HW, consistente en la capacidad de ejecutar más de una tarea al mismo tiempo, se realiza mediante su concurrencia espacial dentro del dispositivo.

El tipo de dispositivos de HW reconfigurable más extendido actualmente es el denominado Field Programable Gate Array (FPGA), dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad se pueden programar después de la fabricación.

La tecnología del HW reconfigurable se sitúa entre la de los procesadores y los circuitos de aplicación específica (ASIC, de *Aplication Specific Circuit*): tiene la flexibilidad de los procesadores y, al mismo tiempo, es capaz de ofrecer muy buenos resultados en términos de rendimiento. En los últimos

años esta flexibilidad se ha incrementado con la aparición de varias características arquitectónicas [CoHa02] tales como la reconfiguración parcial en tiempo de ejecución y el soporte para incluir bloques prediseñados.

Como consecuencia de los avances en la tecnología de fabricación, con 500nm en las primeras familias hasta 65nm en las actuales, el **tamaño** de los dispositivos también se ha incrementado a lo largo de los años, permitiendo que varias tareas concurrentes compartan recursos HW a través de la concurrencia espacial. Esta concurrencia espacial es más eficiente que la multiplexación realizada en el tiempo, usada en sistemas SW convencionales monoprocesador.

La complejidad de los bloques configurables varía mucho de unos sistemas a otros y es lo que determina la **granularidad**. Cuanto más fina sea esta granularidad, más elementos de configuración existen (mayor tamaño del fichero de configuración) y más tiempo se necesita para reconfigurar el dispositivo.

Además, las plataformas reconfigurables actuales (principalmente las FPGAs) han incluido bloques de RAM interna e incluso elementos de grano grueso como multiplicadores o procesadores, que junto a las nuevas características arquitectónicas descritas, permiten aumentar su rendimiento y simplificar el proceso de diseño. Ejemplos de este tipo de plataformas reconfigurables son las familias Virtex2-PRO [Xili07], Virtex-4 [Xili07b] y Virtex-5 [Xili08] de Xilinx, o la familia FPSLIC de Atmel [Atme08].

Este tipo de plataformas heterogéneas permite utilizar únicamente los procesadores mientras los requisitos de rendimiento sean bajos y usar los recursos reconfigurables para cumplir los requisitos de las aplicaciones más exigentes. De esta forma, empleando una política de ahorro de energía que mantenga los elementos que no se estén utilizando en un estado de bajo consumo, se puede conseguir una plataforma con un consumo de energía medio comparable a los procesadores para sistemas empotrados, y que a la vez pueda proporcionar el rendimiento pico que demandan las aplicaciones actuales.

Utilizando la posibilidad de **reconfiguración parcial**, la funcionalidad de los recursos hardware puede modificarse parcialmente en tiempo de ejecución para adaptarse a los requisitos variables de las distintas aplicaciones que se comentarán más adelante.

La reconfiguración parcial dinámica es útil para aquellos sistemas con múltiples funciones que pueden compartir los mismos recursos del dispositivo. En estos sistemas, una sección de la FPGA continúa la operación, mientras otras secciones de la FPGA son deshabilitadas y reconfiguradas para proporcionarles una nueva funcionalidad. Esto es análogo a la situación donde un microprocesador gestiona los cambios de contexto entre procesos software (multitarea SW). En el caso de la reconfiguración parcial en una FPGA, sin embargo, el que se cambia es el hardware en lugar del software, por lo que tenemos una verdadera multitarea HW.

En aquellas aplicaciones que requieren operación continua no es posible realizar reconfiguraciones totales cada vez que hay que actualizar o reemplazar algún módulo, y la reconfiguración parcial es la única alternativa válida. Un posible ejemplo de este tipo de aplicaciones es la visualización de gráficos que utiliza sincronización vertical y horizontal. Debido al entorno en el cual esta aplicación está operando, las señales de los enlaces de radio y video tienen que mantener la operación de modo continuo, pero las señales de procesamiento y el formato de los datos necesitan ser actualizadas y cambian durante la operación. Con la reconfiguración parcial, el sistema puede mantener los enlaces en tiempo real mientras otros módulos dentro de la FPGA son reemplazados y actualizados sobre la marcha (*on the fly*).

Las técnicas ideadas para llevar a cabo la multitarea HW dependen de la **arquitectura interna** de los recursos y del **sistema de reconfiguración** incorporado. La arquitectura interna es en general de dos dimensiones (2D), más o menos homogénea dependiendo de las características de grano consideradas. Sin embargo, las restricciones de los sistemas de configuración en una dimensión (1D, presente en algunas familias como las Virtex y las Virtex-II), llevaron a distintos investigadores a desarrollar metodologías relativamente sencillas para los recursos 2D comerciales,

basados solamente en la gestión 1D. Algunos investigadores han implementado una gestión de HW real en 2D sobre FPGAs comerciales, evitando estas restricciones de reconfiguración por columnas (1D) descritas en [Xili04]. Por otra parte, cuando se dispone de un sistema de reconfiguración 2D, se puede implementar una gestión real con técnicas más complejas e interesantes que permiten una utilización más eficiente de los recursos 2D. Este tipo de técnicas pueden ser implementadas en dispositivos actuales como las Virtex-4 y Virtex-5, ya que disponen de sistema de reconfiguración 2D. Por último, recientemente se han propuesto en [AFGH05] modelos de arquitecturas FPGA tridimensionales (3D) que consideran los avances conseguidos en la tecnología de CIs (Circuitos Integrados) en tres dimensiones. Sin embargo, plantearse multitarea HW sobre este tipo de arquitectura hipotética parece todavía demasiado especulativo.

Un aspecto importante que hay que resaltar es que no existe ningún soporte comercial específico para hardware reconfigurable que permita gestionar de forma sencilla la creación de tareas de forma dinámica, ni la sincronización y comunicación de estas tareas entre sí y con el resto de elementos del sistema. Aparentemente, la multitarea HW no interesa aún desde el punto de vista comercial. Sin embargo resulta evidente que existen en la industria numerosos campos de aplicación que podrían beneficiarse del uso de este tipo de técnicas.

La evolución de la industria del semiconductor, y la de los dispositivos reprogramables en particular, están marcadas en la actualidad por la demanda del mercado del ocio (electrónica de consumo) y las aplicaciones en red (*networking*). La electrónica de consumo representa la mayor demanda del mercado semiconductor y engloba entre otros a teléfonos con videoconferencia, TV LCD y plasma, videoconsolas, cámaras y grabadoras digitales. Estas aplicaciones son ideales para el uso de HW reconfigurable ya que se dispone de bloques prediseñados con funcionalidades específicas como codificadores/decodificadores de imagen, audio y video, controladores de LCD, de teclado, de VGA, etc. Todas estas facilidades aceleran la fase de desarrollo porque sólo hay que integrar los prediseños y adaptarlos a la aplicación disponiendo rápidamente de un prototipo para pruebas.

Para implementar una funcionalidad concreta, la mejor solución tanto en rendimiento como en consumo de energía durante mucho tiempo ha sido incluir en el sistema un circuito hardware específico ASIC diseñado para ejecutarla de forma óptima. Sin embargo, unos tiempos de mercado cada vez más cortos y el continuo aumento del número de aplicaciones que deben incluirse en este tipo de sistemas provocan que, frecuentemente, no sea posible utilizar ASICs para conseguir la capacidad de computación necesaria para todos los algoritmos utilizados por la mayoría de aplicaciones. Por ejemplo, los teléfonos móviles han comenzado a incluir codificadores y decodificadores de vídeo, imagen y audio, aplicaciones con gráficos en tres dimensiones y soporte para diversos estándares de conexión inalámbrica. En un ordenador personal estas aplicaciones se optimizan incluyendo tarjetas de sonido, de vídeo, aceleradores gráficos y tarjetas para conexión inalámbrica, que incluyen ASICs para proporcionar el rendimiento necesario. Sin embargo, en los sistemas empotrados (sistema informático de uso específico construido dentro de un dispositivo mayor) no hay suficiente espacio para incluir estos elementos y además no permiten que se modifique el hardware a posteriori, por lo que las FPGAs combinadas con el uso de bloques prediseñados es una buena solución.

Dentro de las aplicaciones en red, el incremento en el tráfico en las redes de interconexión ha propiciado el uso de FPGAs en productos que deben manejar paquetes de datos a muy alta velocidad y proporcionar servicios de seguridad y monitorización de la red, análisis del tráfico, etc.

Las FPGAs encuentran también aplicaciones en aquellas áreas que hacen uso o explotan el paralelismo masivo ofrecido por su arquitectura, un buen ejemplo son los rompedores de código, en particular algoritmos de fuerza bruta (*brute force attack*). Los factores del mercado y características que aportan las FPGAs han propiciado que el uso intensivo de estos dispositivos haya desplazado al empleo de ASICs y DSPs.

Otro ejemplo que se puede citar es el procesamiento de imagen, donde tradicionalmente se han empleado procesadores de propósito general, DSPs y ASIC. En la actualidad los dos primeros no pueden soportar la demanda de mayores resoluciones y refresco de imagen, mientras que los ASIC no se

adaptan a los tiempos de salida al mercado actuales. Este factor, combinado con el abaratamiento de las FPGAs, ha desplazado en parte el uso de todos los dispositivos mencionados anteriormente en este campo.

También en la industria de la automoción las FPGAs se emplean de forma masiva en muchos sistemas, como el de gestión del motor, de seguridad (p.e. control de airbags), de control de frenos y de procesamiento de imagen, que pueden ser muy críticos y necesitan gestión en tiempo real.

En otros campos donde las condiciones del entorno son críticas, que tienen requisitos de procesamiento en tiempo real y fuertes restricciones de peso, tamaño y potencia, las FPGAs también se están empleando de manera masiva. Un ejemplo de este tipo de industria son las aplicaciones espaciales, militares y aviones no tripulados.

En el campo de la bioinformática las FPGAs se utilizan para ejecutar de forma masivamente paralela los algoritmos más utilizados, como los empleados para determinar el grado de semejanza entre dos secuencias de ADN o de proteínas.

La industria sísmica demanda un consumo insaciable de procesamiento de datos. Hasta hace poco se utilizaban grupos (*clusters*) de estaciones de trabajo con capacidades de procesamiento de Gigaflops, pero como estaban basadas en procesadores de propósito general, no eran óptimas para entregar potencias del orden de Teraflops necesarias en la actualidad. La FPGA es un candidato idóneo para ejecución de algoritmos de cálculo matemático intensivo que convierten datos sin tratar y se ejecutan billones de veces sobre el mismo conjunto de datos.

El hardware dinámicamente reconfigurable tiene por tanto las características ideales para resolver los requisitos de estos tipos de aplicaciones críticas ya que, por un lado, puede alcanzar el rendimiento necesario al permitir implementar circuitos que aprovechen al máximo el paralelismo de cada tarea de la aplicación; y por otro, un mismo recurso de hardware reconfigurable puede utilizarse como acelerador para un número de aplicaciones virtualmente ilimitado (en la práctica el número de

aplicaciones que soporte únicamente está limitado por el espacio de memoria asignado a almacenar las configuraciones susceptibles de ser cargadas).

Para permitir multitarea HW y explotar al máximo sus posibilidades en sistemas que incorporan elementos de hardware reconfigurable, la funcionalidad del Sistema Operativo (SO) se debe extender para gestionar este tipo de recursos. En [WiKe02] se muestran algunos de los principales problemas que un SO debe resolver para llevar a cabo una multitarea HW real. Entre ellos, los problemas relacionados con la planificación de tareas, la selección de una ubicación para el mapa de bits de la tarea reubicable, la gestión de E/S de la tarea, las técnicas para reducir la latencia de la carga de configuración o la gestión de la defragmentación a través de la reubicación de tareas, deben de ser tratados. Otros problemas interesantes, especialmente aquellos relacionados con las características de grano grueso tales como bloques RAM, o la relación entre los elementos reconfigurables y los procesadores empotrados, caen más bien dentro del dominio de otras áreas de investigación tales como el codiseño HW/SW.

Cuando se trabaja con sistemas empotrados, un sistema operativo del tipo RTOS (*Real Time Operating Systems*), especialmente diseñado para sistemas empotrados, debería ser el encargado de proporcionar este soporte. Sin embargo ningún RTOS comercial actual es capaz de gestionar los recursos de HW dinámicamente reconfigurable, por lo que el diseñador debe crear un módulo que realice esta gestión. Esta falta de soporte ha provocado que prácticamente ninguna plataforma comercial actual utilice la reconfiguración parcial en tiempo de ejecución, aunque los recursos dinámicamente reconfigurables comerciales soporten este tipo de reconfiguración desde finales de la década de los noventa.

Entre todos los problemas que debe resolver un módulo gestor de HW reconfigurable, en este trabajo de investigación se van a tratar los siguientes: la ubicación de tareas, la fragmentación del espacio libre y las técnicas que se pueden utilizar para reducir dicha fragmentación.

La **ubicación de tareas** consiste en decidir la posición donde escribir el mapa de bits parcial de una tarea nueva en la FPGA. Esta ubicación de tarea

asigna unos recursos HW concretos para la ejecución de una tarea, que no serán liberados hasta que no finalice su ejecución (o el Gestor de HW decida interrumpirla). La mayoría de los investigadores tratan este problema centrándose en la gestión de recursos de grano fino homogéneos y las soluciones propuestas para la ubicación de tareas están muy relacionadas con el modo en el que se describe el espacio libre.

Para mantener la información que describe el área libre disponible del dispositivo se necesita una estructura de datos, y el algoritmo de ubicación debe buscar y elegir el mejor sitio para situar la tarea entrante que está demandando recursos del dispositivo. Este algoritmo debe intentar usar el área reconfigurable tan eficientemente como sea posible. La bondad del algoritmo se medirá con los parámetros utilizados en los apartados de resultados experimentales (volumen de cálculo rechazado, el porcentaje de utilización de la FPGA y el tiempo de ejecución del algoritmo) y también por una medida indirecta como puede ser el nivel de **fragmentación de los recursos**.

Por otra parte, incluso con buenas heurísticas de ubicación de tareas, la fragmentación del espacio disponible es inevitable a medida que las tareas entran y salen de la FPGA. Como el estado de fragmentación es una medida indirecta de la probabilidad de encontrar una posición adecuada para insertar una nueva tarea en la FPGA en un futuro próximo, se necesitan por tanto medidas que permitan **defragmentar** los recursos disponibles para que puedan aprovecharse mejor. Este tipo de medidas además pueden realizarse de forma preventiva.

A continuación se desarrollará con mayor detalle la evolución del HW reconfigurable y se revisarán en profundidad los aspectos y problemas que plantea la multitarea HW.

1.1 Introducción al HW reconfigurable

La revolución digital que empezó en los años 80 con la aparición del computador personal continuó durante diez años hasta llegar a superar el mercado analógico. Ahora en este nuevo siglo, ha resurgido una nueva revolución digital caracterizada por productos digitales de consumo orientado al ocio y las aplicaciones en red. La industria del semiconductor también se ha transformado a lo largo de este proceso, orientándose en parte hacia tecnologías programables en campo (por el diseñador) y dispositivos reconfigurables.

Esta industria ha tenido ciclos alternativos de estandarización y personalización según predice el Dr. Tsugio Makimoto en [Maki00], donde se descubrían ciertas reglas de regularidad en las tendencias del mercado del semiconductor. La característica principal de esta regularidad residía en la aparición de ciclos repetitivos a largo plazo semejantes al movimiento de un péndulo de reloj. La explicación era que cuando aparecen un gran número de nuevas tecnologías, tales como dispositivos, arquitecturas o software, la industria del semiconductor se mueve hacia una estandarización. Entonces aparecen factores que frenan el movimiento, tales como necesidades de diferenciación de un producto y de valor añadido. En la otra dirección, los progresos en la automatización del diseño y avances en la tecnología tales como CAM (*Computer Aided Manufacturing*) y CAT (*Computer Aided Test*) cambian la tendencia hacia una personalización.

Como se puede ver en la figura 1.1, en los años 70, una onda de equipos de consumo analógico, aparatos de video y televisión, arrasó en el mundo tecnológico. Seguido a esto, en los 80 apareció la primera revolución digital, con el advenimiento de los computadores personales, y en los 90 la onda analógica desapareció. La segunda revolución digital empezó en los 90 y continúa en el presente, caracterizada por productos de consumo y aplicaciones en red.

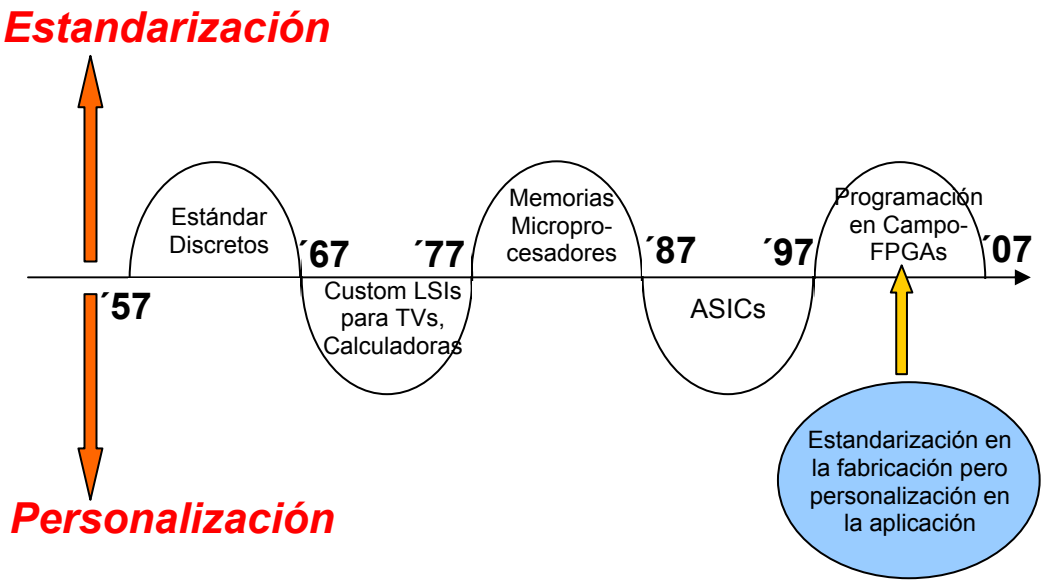


Figura 1.1. Onda de Makimoto.

De acuerdo a esas predicciones, la tecnología de los dispositivos programables se convertiría en la tecnología líder en un ciclo de diez años a partir de 1997. Las predicciones de Dr. Makimoto se han cumplido y en la actualidad los dispositivos de lógica programable se han situado como un mercado líder de la industria del semiconductor [Hand04].

La denominación de dispositivos programables engloba a todos aquellos circuitos digitales cuya función puede definir el diseñador mediante la programación e interconexión de los elementos que los forman. Se clasifican habitualmente en tres tipos: Dispositivos Lógicos Programables Simples (SPLD), Dispositivos Lógicos Programables Complejos (CPLD) y los Conjuntos Programables de Puertas (FPGAs). Este tipo de dispositivos son usados para construir desde simples circuitos digitales hasta sistemas completos (SOC, de *System On Chip*). La principal diferencia con los ASIC, es que estos tienen su función fijada desde su fabricación, mientras que en los PLDs y las FPGAs la función está sin definir en fabricación y antes de ser usada debe ser programada por el diseñador.

Un precursor de los dispositivos programables SPLD es el chip de memoria de sólo lectura (ROM, de *Read Only Memory*) que se utilizaba para crear

funciones de lógica combinacional arbitrarias con un número determinado de entradas. Sus principales desventajas fueron que consumía más que su equivalente de lógica dedicada, realizaba un uso ineficiente (sólo se usa una fracción de la capacidad del dispositivo) y no contenía elementos de lógica secuencial. Después aparecerían las memorias reprogramables EPROM y EEPROM como evolución de las primeras ROM, que pueden ser borradas con luz UV y de modo eléctrico respectivamente.

Existen varios tipos de SPLDs disponibles, pero el primer dispositivo programable real fue la PAL (Array de Lógica Programable) que apareció en 1978 y reemplazaría docenas de dispositivos lógicos discretos SSI (*Small Scale of Integration*) que dejaron paso a las PAL en muchos productos, como minicomputadores. Las primeras PAL, que fueron descritas como MSI (*Medium Scale of Integration*), constaban de dos planos o niveles de puertas para configurar cualquier función lógica en forma de suma de productos, con un plano AND programable, el plano OR fijo, y elementos programables (fusibles) que conectan las entradas a las puertas AND. Entre sus principales ventajas cabe destacar que era simple de fabricar y que tenía retardos pequeños y predecibles. Una innovación de la PAL fue la Matriz Lógica Genérica (GAL, de *Generic Array Logic*), que apareció en 1985. Este dispositivo tiene las mismas propiedades lógicas que la PAL, pero puede ser borrada y reprogramada, siendo muy útil en la fase de prototipado de un diseño, cuando un fallo en la lógica puede ser corregido por reprogramación. Otro tipo de dispositivo SPLD es la PLA, muy parecida a la PAL pero con el plano OR también programable.

Los dispositivos SPLD evolucionan hacia los CPLD (PLD Compleja), que contienen múltiples PAL unidas con interconexiones programables en un mismo dispositivo y permiten implementar más ecuaciones lógicas o diseños más complejos (reemplaza cientos de dispositivos SSI).

Mientras que las PAL evolucionan hacia las GAL y los CPLD, surge una corriente de desarrollo separada con una estructura diferente basada en un *array* de lógica. La primera FPGA aparece en 1985 y se usa para implementar cualquier tipo de diseño HW, además de prototipos antes de fabricar un ASIC. Su arquitectura consta de 3 elementos principales: bloques

lógicos, bloques E/S y la interconexión programable que conecta bloques de lógica entre sí y con los de E/S.

Las principales diferencias entre las CPLDs y las FPGAs derivan de su arquitectura y tecnología de fabricación. En cuanto al interconexionado las CPLD lo tienen de tipo *crossbar* y cada salida es directamente interconectable a cada entrada a través de 1 o 2 conmutadores, por lo que sus retardos son menores y más predecibles. En las FPGAs el interconexionado es segmentado, por lo que las conexiones entre los bloques de lógica pasan típicamente a través de varios conmutadores. En cuanto al tamaño, como en las FPGAs se incluyen mayor número de elementos lógicos, aunque son más sencillos que los de una CPLD, permiten implementar diseños más complejos y de mayor tamaño.

Las FPGAs y los CPLDs son buenas opciones para un mismo diseño cuando no es demasiado grande y algunas veces la decisión sobre una u otra es más económica que técnica, o puede depender de la preferencia personal o experiencia del diseñador. Sin embargo como las FPGAs tienen mayor densidad de puertas y menor coste que los CPLDs, al final se opta por los dispositivos del tipo FPGA ya que permiten implementar una mayor variedad de sistemas dentro de un único dispositivo.

Aunque las FPGAs todavía adolecen de problemas relacionados con la velocidad y el consumo, estos dispositivos se consideran como una alternativa flexible y barata frente a los ASIC y en la actualidad su densidad ha sobrepasado la barrera de los 10 millones de puertas.

Por otra parte se espera que el rendimiento de las FPGAs se incremente en la misma proporción en los próximos años. Esta mejora es debida a la combinación de la reducción en el tamaño de los transistores, incremento en las velocidades de reloj, mejoras arquitectónicas y disponibilidad de mejores herramientas de diseño CAD.

Entre las ventajas que aportan los sistemas basados en FPGAs se puede citar que son más aptos para desarrollos en pequeñas cantidades, aunque actualmente comienzan a serlo también para grandes cantidades. Al ser reconfigurables, con las FPGAs se pueden corregir errores en productos ya

instalados. Las herramientas de desarrollo son eficaces y accesibles, tienen tiempos de desarrollo más cortos y costes de desarrollo muchos menores.

Como desventajas frente a los ASICs se pueden considerar la menor protección de los diseños frente a copias (generalmente es más fácil de copiar en las FPGAs), su menor velocidad de procesamiento y su mayor consumo de energía, y el hecho de que los ASICs siguen siendo más baratos para cantidades muy grandes y tienen mayor densidad de funcionalidad.

Si tenemos en cuenta el factor temporal, muy importante en las principales aplicaciones de la lógica reconfigurable, como son los tiempos de salida al mercado y el tiempo de vida útil, éstos se han acortado considerablemente (en algunos casos el ciclo de vida de un producto ha disminuido de 5 años a 1), y en muchos casos las FPGAs son las únicas alternativas válidas.

1.2 Evolución de las características de las FPGAs

Dado que este trabajo de investigación se centra en la gestión de dispositivos dinámicamente reconfigurables para permitir multitarea HW, en este apartado se revisará la evolución de los dispositivos reconfigurables de tipo FPGA desde el punto de vista del **incremento del tamaño, evolución arquitectónica** (tipo de recursos), métodos y estructura de **reconfiguración** (reconfiguración parcial)**,** soluciones para la **comunicación** entre tareas y evolución de las **herramientas** de diseño. Para ello nos vamos a centrar en la sucesión de familias de uno de los fabricantes más importantes de este tipo de dispositivos como es Xilinx.

La primera familia de Xilinx, la XC2000, que aparece en 1985, presenta solamente dos dispositivos, con 1200 y 1800 puertas de sistema. La segunda generación (1988), la XC3000, amplía a cinco los tipos de dispositivo [Xili98], con capacidades desde 1500 hasta 7500 puertas. La tercera generación, la XC4000 [Xili99] aparece en 1992 e incluye RAM dentro del dispositivo. Es la primera en llegar a la barrera de 20.000 puertas y amplia los modos de

configuración disponibles: puede ser configurada externamente en serie o paralelo (modo *master*), o accediendo directamente a la FPGA (modo *slave* o periférico). Después aparecerá en 1996 la XC6200 [Xili96], que aunque sólo se comercializó durante un corto periodo, es importante por ser la primera en ofrecer la posibilidad de reconfiguración parcial e incorporar un sistema de reconfiguración 2D, llegando a densidades de 100.000 puertas. Posteriormente aparecen las familias de Virtex, donde los factores de escala mantienen el progreso de la Ley de Moore (capacidad de la industria para doblar el número de transistores de un chip cada dos años).

Se puede observar en la figura 1.2.a que cada dos años se duplican las densidades de los dispositivos (medidos en celdas lógicas), salvo la transición de Virtex-II a Virtex-IIPro, que aunque sigue aumentando de tamaño, este aumento no se refleja en un aumento del número de recursos reconfigurables porque incorporan elementos de grano grueso (PowerPc405 y *transceivers* (transmisor/receptor) de comunicaciones de alta velocidad) que no se consideran en la cuenta. Todos estos datos se han extraído de los documentos [Xili07b] [Xili07c] [Xili07d] [Xili08] y [Xili08b].

De un modo similar, los **mapas de bits de configuración** han crecido en la misma proporción que la densidad de los dispositivos, en la actualidad superando los 80Mbits, mientras que la velocidad de transferencia de los ficheros de configuración no ha evolucionado en la misma proporción. Para la familia Virtex y Virtex-II la frecuencia es de 50MHz, mientras que para las Virtex-4 y 5 es de 100MHz.

Una ventaja añadida de la reconfiguración parcial es que para realizar una pequeña actualización del diseño no es necesario enviar el fichero completo, sino pequeños mapas de bits parciales. Si consideramos el aumento progresivo en el tamaño de los ficheros de configuración, sería preferible una modificación parcial de una pequeña parte en lugar de una reconfiguración total, especialmente en sistemas en los que una parte debe seguir operando igual. El aumento del tamaño de los ficheros de configuración se muestra en la figura 1.2.b para algunas de las familias Virtex de Xilinx y se puede comprobar la relación directa que hay entre el tamaño de dispositivo y el fichero de configuración.

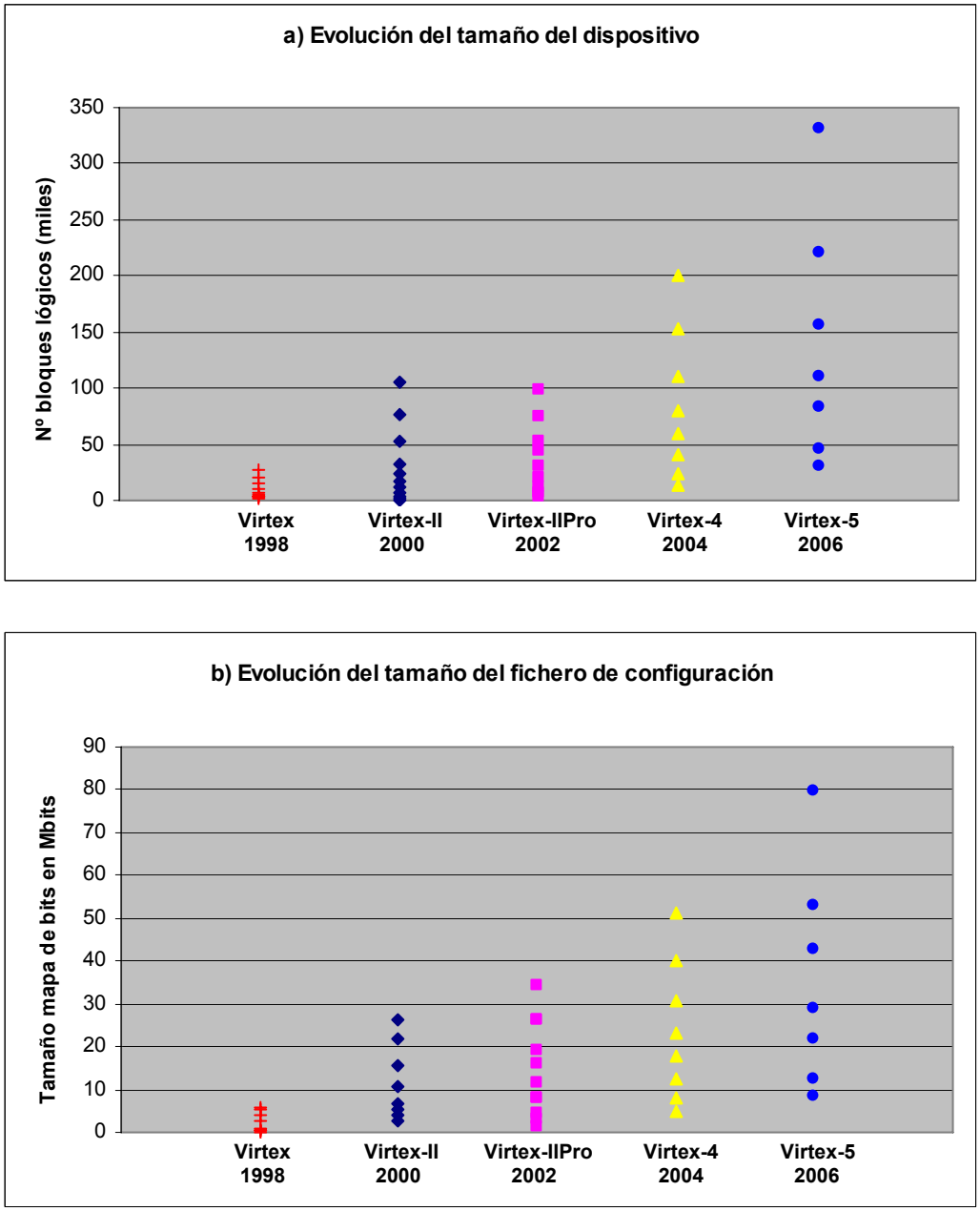


Figura 1.2. Evolución del tamaño del dispositivo (a) y de los ficheros de configuración (b) en familias Virtex.

Desde el punto de vista **arquitectónico**, en los primeros modelos impera la homogeneidad y el predominio de grano fino. A medida que las tecnologías de fabricación permiten incrementar la densidad de los dispositivos, empieza a aparecer la heterogeneidad que va unida al incremento de tamaño y a la presencia de grano grueso.

Esta presencia de grano grueso empieza a partir de las familias Virtex, mientras que la diferenciación y especialización dentro de la misma familia aparece a partir de las Virtex-4, en la que se ofrecen las variantes LX (predominio de lógica), SX (dispone de bloques especializados en procesamiento digital de señal) y la familia FX (dispone de procesadores PowerPC empotrados dentro de la FPGA). Toda esta evolución coincide de nuevo con las predicciones realizadas en la curva de Makimoto (figura 1.1), en la que después de una estandarización aparece una diferenciación de producto.

Por otra parte, si exceptuamos el caso excepcional de la XC6200, cuando aparece la reconfiguración dinámica en las primeras familias de FPGAs (dispositivos Virtex, Virtex-II y Virtex-II Pro de Xilinx), sólo se ofrecían modelos con **arquitectura de configuración de una dimensión (1D)**. La arquitectura de configuración de la familia Virtex se describe en la nota de aplicación [Xili04] de Xilinx, y es esencialmente la misma para las tres series anteriores.

Esta arquitectura 1D admite **reconfiguración parcial**, y los datos de configuración se almacenan en SRAM que puede ser escrita o leída sin detener el funcionamiento del dispositivo. Como muestra la figura 1.3, todos los dispositivos Virtex-II/Pro están divididos en cuatro cuadrantes, donde cada cuadrante soporta hasta ocho relojes. La unidad mínima que puede ser leída o escrita es un ***frame***, que se expande a la altura total del dispositivo, con un ancho de un bit en el caso de las Virtex.

Este tipo de arquitectura de reconfiguración parece sugerir una gestión del área reconfigurable en 1D, por columnas. Sin embargo, es posible que una tarea pueda tener una posición y un tamaño arbitrarios, teniendo en cuenta que en ese caso los datos de configuración y estado escritos en las mismas columnas y que son ajenos a la tarea, se deben sobrescribir con los mismos valores de estado y configuración existentes. De este modo se puede realizar una gestión 2D sobre una arquitectura de reconfiguración 1D.

La figura 1.3 muestra la arquitectura de configuración de la Virtex-II, donde aparece la mínima porción reconfigurable que corresponde a un *frame* de

altura igual al dispositivo y ancho un bit. También aparecen los elementos principales de la FPGA: los bloques de E/S (IOBs), bloques lógicos configurables (CLBs), y los elementos de grano grueso BlockRAM y Multiplicadores.

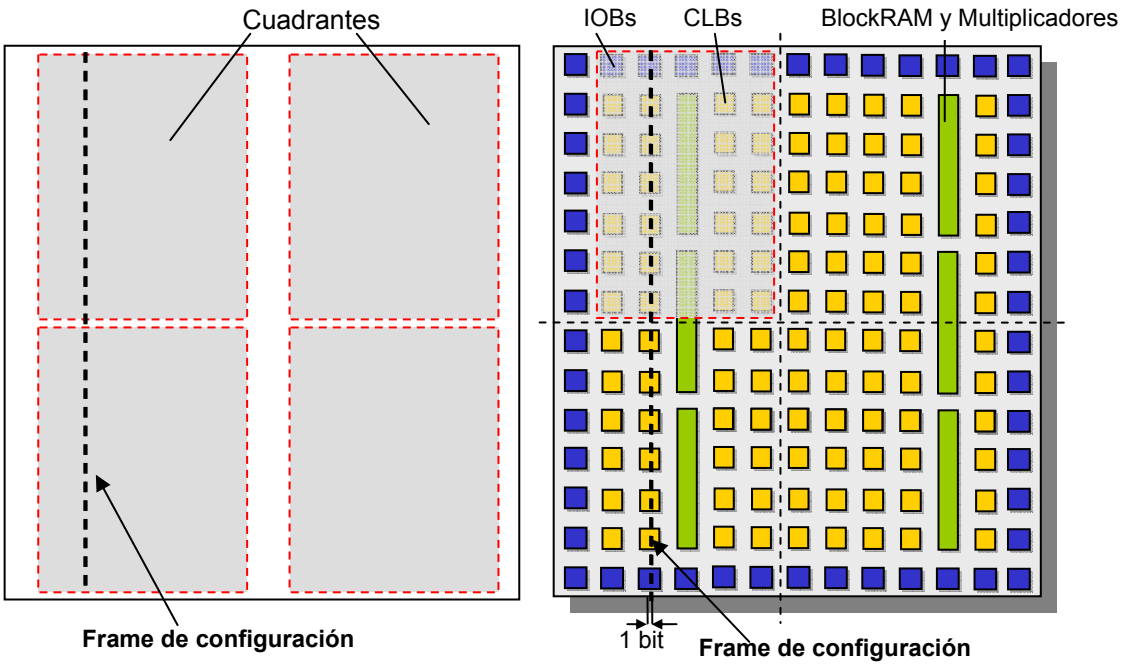


Figura 1.3. Arquitectura de configuración de Virtex-II.

Sin embargo, las últimas familias de Virtex (Virtex 4 y 5) marcan un cambio significativo en la disposición de recursos (*layout*) respecto de familias anteriores. Como muestra la figura 1.4, ahora los *pads* de E/S están dispuestos en columnas, como el resto de recursos, en lugar de en anillo en la periferia del dispositivo (Virtex-II/Pro), que se podía observar en la figura 1.3.

En la arquitectura de configuración de la Virtex-4 también hay importantes diferencias respecto de familias anteriores. Ahora el dispositivo está dividido en regiones de reloj (en lugar de cuadrantes) donde cada región se expande a una altura fija de 16 CLBs (20 CLBs en la Virtex-5).

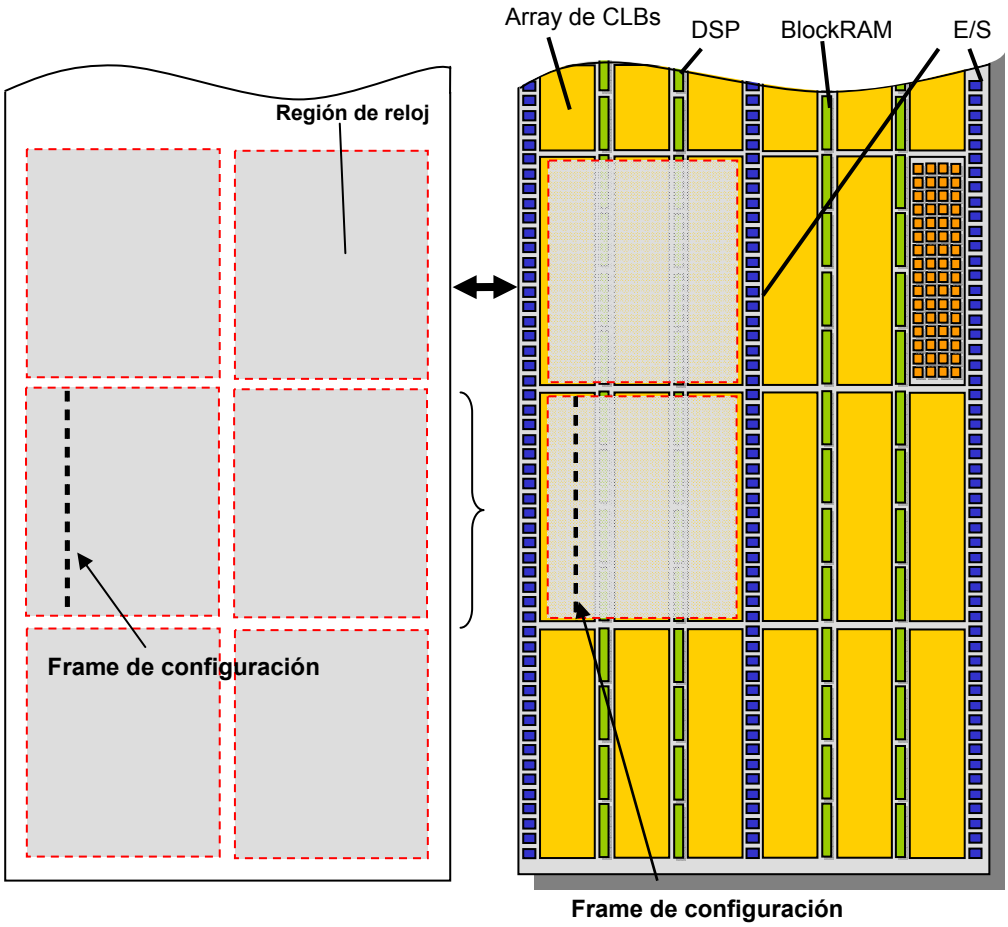


Figura 1.4. Arquitectura de configuración de Virtex-4.

Aunque la unidad mínima de reconfiguración es todavía un frame, **dicho frame ahora se expande a la altura de una región de reloj, en lugar de la altura total del dispositivo**. Teniendo en cuenta esta característica y si hacemos coincidir una región de reloj con el tamaño mínimo de región reconfigurable, es decir, la tarea más pequeña posible para un dispositivo pequeño como la XC4FX12, tendríamos una organización de la arquitectura en 2D de 2*8 regiones reconfigurables, mientras que para la XC4FX140 sería de 2*12 regiones reconfigurables. Este cambio es muy importante ya que permite y simplifica la gestión real en 2D.

Aunque sería posible reconfigurar tamaños menores de tarea que una región de reloj, esta restricción tiene una ventaja adicional: cada módulo reconfigurable puede disponer de su propia región de reloj independiente del

resto de módulos, y los módulos de gestión de reloj (DCMs, de *Digital Clock Manager*) se pueden reconfigurar para ajustar la frecuencia de reloj de cada módulo reconfigurable de manera independiente.

En cuanto a la **comunicación de tareas** o módulos reconfigurables, para las familias Virtex-II/Pro se tenía que realizar a través del bus macro triestado descrito en [Xili04b]. Sin embargo, en las familias posteriores como las Virtex 4 y 5, como no tienen líneas ni *buffers* triestado [Xili02], se realiza una comunicación basada en Slices, denominada *Slice Macro*. En el apéndice A-2, se encuentra información más detallada de la comunicación entre tareas, así como de algunas implementaciones prácticas realizadas en familias Virtex.

En cuanto a las **herramientas de diseño**, en la primera época (desde 1984 hasta 1992) se disponía de una versión del software *XACT Development System* de Xilinx que soportaba las primeras familias XC2000 y XC3000. Permitía realizar todas las fases del diseño (entrada con un editor del diseño, simulación, rutado y asignación), aunque no de una forma totalmente integrada en el mismo entorno. Además, este software no disponía de ninguna herramienta de asignación y rutado automatizados y consistía principalmente en el *Xilinx Design Editor* para la entrada del diseño mientras que el rutado y asignación se realizaban manualmente. En 1986 aparece la primera versión del programa *Automatic Place and Route* y posteriormente aparece una nueva versión del *XACT* (*XACT Step*, desde 1990 hasta 1996), que soporta los dispositivos XC3000 y XC4000.

En la siguiente época, desde 1996 hasta 2002, aparece un entorno de diseño integrado denominado ISE, que soporta todos los dispositivos XC4000 y las familias Spartan y Virtex. Versiones posteriores denominadas ISE Foundation, desde 2002 hasta ahora, soportan desde Spartan-II hasta todas las familias de Virtex.

Sin embargo, como comentario general se podría destacar la falta de soporte oficial por parte del fabricante para poder realizar la reconfiguración dinámica en estos dispositivos (especialmente en Virtex-4 y Virtex-5), ya que hasta el momento no hay herramientas comerciales ni ninguna nota de

aplicación proporcionada por Xilinx a este respecto. Cuando se empezaron a usar las FPGAs, la reconfiguración dinámica no era posible y todos los aspectos anteriores no eran tan importantes porque la configuración sólo se cargaba en el arranque, pero ahora todas estas características mencionadas adquieren gran importancia.

Con todo lo anterior queda claro que en la actualidad resulta plenamente factible y defendible una gestión de multitarea HW en 2D, aunque las herramientas todavía no lo soporten adecuadamente.

1.3 Problemas asociados a la gestión de multitarea Hw

La necesidad de un gestor de recursos hardware (HW) dinámicamente reconfigurables con funcionalidad semejante a la de un SO tradicional se propuso ya en [Breb96] o en [MeLJ98], donde se identificaban los problemas básicos a resolver para permitir multitarea HW y la forma de extender la funcionalidad de un SO. Esta cuestión ha sido ya planteada por varios investigadores, como en [WiKe01] y [WiKe02], donde se hace una minuciosa revisión de todos los problemas abiertos en este campo, y han llegado a implementarse en algunos sistemas reales, como el descrito en [BoSr00], que incorporan en un sistema HW/SW multitarea (si bien no es exactamente de propósito general) estrategias que abordan algunos de ellos. En particular pueden destacarse los problemas relacionados con la asignación de una porción concreta del dispositivo de HW reconfigurable para la ejecución de una tarea dada, la posibilidad de reubicar la tarea dentro del dispositivo y el problema de la fragmentación del espacio libre dentro del dispositivo.

A continuación se revisarán las principales funciones que sería preciso realizar para gestionar por parte del SO unos recursos HW dinámicamente reconfigurables de gran capacidad.

a. Compilación de las tareas a código HW reubicable. En primer lugar, las tareas que puedan ser susceptibles de ejecutarse en HW deberán haber sido compiladas para el HW objetivo, dando como resultado un código HW reubicable que pueda ser cargado donde se disponga de espacio libre. Sin embargo, para que las tareas puedan reubicarse en cualquier posición libre, el sistema que gestione el HWDR debe tener una visión de que los recursos de HW son homogéneos. Existen algunas implementaciones prácticas donde se manipula el mapa de bits inicial de una tarea para poder variar su ubicación mediante la modificación de la dirección de columna, como las herramientas REPLICA Filter [KLPR05] para arquitecturas de grano fino totalmente homogéneas como en las Virtex, y REPLICA2Pro [KaPo06] para arquitecturas heterogéneas que incorporan elementos de grano grueso como la familia Virtex-II/Pro.

Además, si se está contemplando la posibilidad de ejecutar una tarea tanto en SW como en HW dependiendo de la disponibilidad de los recursos, deberían haber sido doblemente compiladas, tanto a SW como a HW reconfigurable dinámicamente, sin embargo su gestión cae en el dominio de otras áreas de investigación tales como el codiseño HW/SW.

b. Gestión de la información sobre los recursos HW disponibles en cada momento. El SO debe disponer en cada momento de la información suficiente sobre el estado de los recursos HW como para poder tomar las decisiones relativas a su gestión. Esta información debe ser almacenada en estructuras de datos que deberían cumplir unas características básicas: flexibilidad, cómoda actualización, búsqueda rápida de la información y precisión para describir el estado de los recursos. La actualización de esta información se realizará normalmente cuando se asigna una porción de los recursos HW de la FPGA a una tarea para su ejecución y cuando se detecta la finalización de una tarea, y conllevará la actualización de la información disponible por el SO. Sin embargo, es posible tener en cuenta otro tipo de consideraciones, como por ejemplo no liberar el HW ocupado por una tarea saliente si se prevé que puede ser vuelta a cargar y si no es imprescindible reutilizar el espacio de HW reconfigurable que ocupa. En esta aproximación,

la información del sistema sobre la ocupación del HW reconfigurable debería distinguir entre recursos libres, usados, y ocupados pero no usados. Esta distinción, que puede permitir eliminar la necesidad de la reconfiguración en algunos casos, sin embargo complica notablemente una posible tarea de defragmentación.

c. Planificación de tareas HW. En el contexto de un SO multitarea, si asumimos que las tareas HW no van a tener dependencias de datos entre sí, la planificación de las tareas HW deberá basarse en criterios como secuenciamiento, disponibilidad de recursos (una tarea puede adelantar su ejecución sobre sus predecesoras si estas no caben en el HW disponible en ese momento), prioridad relativa de las tareas, restricciones temporales propias de cada tarea (*timeouts*), penalización en el rendimiento total en caso de no ejecutarse la tarea, etc.

En varios contextos o aplicaciones puede ser necesario suspender total o temporalmente la ejecución de una tarea. Como ejemplo pueden considerarse entornos donde una tarea prioritaria puede suspender la ejecución de otra de baja prioridad y así poder usar los recursos HW que estaban ocupados. El SO debe poder entonces reiniciar la tarea suspendida, o reanudar su ejecución en el punto en que se interrumpió. Otro ejemplo son los procesos de defragmentacion, donde las tareas se reubican a otras posiciones con el objetivo de reducir el nivel de fragmentación del espacio libre. En este tipo de contexto es necesario disponer de una herramienta que permita grabar/recuperar el estado de las tareas que son suspendidas o reubicadas, para poder reanudar en el futuro su ejecución en el mismo punto donde se pararon. Un ejemplo de este tipo de herramientas se puede encontrar en [KaPo05] y [KaPo06].

d. Ubicación de tareas. Partimos, como ya se ha comentado anteriormente, de que el código HW de la tarea es de naturaleza reubicable, por lo que podemos asignar a la tarea la ubicación que más convenga para mejorar el rendimiento del sistema.

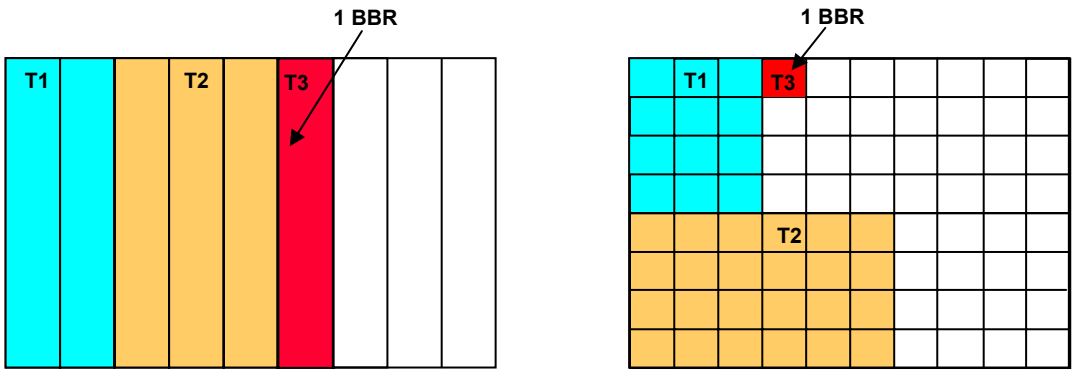
La complejidad del proceso de asignación de recursos HW para la ubicación de tareas, así como la estructura de representación usada, pueden depender considerablemente del modelo de arquitectura interna del HWDR. Dentro del modelo de arquitectura interna los factores principales son: el modelo de reconfiguración 1D ó 2D, el tipo (fijo o arbitrario) y tamaño de la partición, y la forma de las tareas. De este modo, todo el espacio de recursos reconfigurables se organiza y divide en un número de regiones no solapadas (particiones) que se pueden asignar a una tarea para su ubicación.

Para un **modelo de configuración en 1D** como el de las Virtex [Xili04], en el que el elemento básico reconfigurable dinámicamente es una columna, la asignación de una ubicación para la tarea debe gestionar el espacio libre únicamente en una dimensión (por columnas), tomando como unidad básica un tamaño de columna prefijado (variable para cada modelo concreto de la familia). En este caso, las técnicas a utilizar se asemejarían bastante a las empleadas para gestión de memoria.

Para **modelos reconfigurables dinámicamente en 2D**, como la XC6200 [Xili96] y las Virtex 4 [Xili07c] y 5 [Xili08b], el proceso de asignación de recursos resulta mucho más complejo e interesante. En este modelo 2D, partiendo de una región reconfigurable bidimensional, el HW a gestionar puede verse como un teselado regular, sobre el que pueden ubicarse las tareas HW.

Un factor determinante para conseguir que se optimice el uso de los recursos reconfigurables es definir el tamaño adecuado para la unidad básica de gestión, que determinará además el tamaño de la tarea más pequeña posible. Llamaremos a esta unidad en adelante Bloque Básico Reconfigurable (**BBR**). El tamaño y características del BBR influye en muchos aspectos del problema, y a la vez está condicionado por tres parámetros: el tipo de arquitectura, la limitación de la fragmentación generada y las necesidades de E/S de las tareas. En cuanto a la dependencia del tipo de arquitectura interna del dispositivo, el BBR será una columna en el modelo 1D y un rectángulo en el 2D. Además el BBR debe tener suficiente tamaño para que la tarea mínima incorpore un interfaz de E/S. La figura 1.5 muestra ejemplos de arquitecturas en 1D (a) y 2D (b) y tareas de tamaño múltiplo del

BBR, donde en ambos casos T_3 representa una tarea de tamaño mínimo, es decir de un único BBR.



a) 1D b) 2D

Figura 1.5. Ubicación de tareas en HWDR con arquitecturas de una (a) y dos dimensiones (b).

La ubicación de las tareas depende también de la forma en que se gestionan las particiones en las que se van a ubicar.

Si se contemplan sólo **particiones de tamaño fijo**, la gestión es similar a la de la memoria paginada y resulta relativamente simple, tanto en una como en dos dimensiones (ver figura 1.6). Además del tipo de organización interna de la FPGA, hay que considerar la posición donde se va a ubicar la tarea dentro de la partición. La información necesaria sobre el espacio libre es un único bit asociado a cada partición. Además para este tipo de partición simple, los algoritmos de asignación son sencillos: para la selección de la partición se pueden emplear estrategias simples como por ejemplo primera partición libre, o partición más aproximada en tamaño y para la ubicación dentro de la partición es irrelevante (p.e.: esquina inferior izquierda).

Pueden plantear problemas las tareas de tamaño grande, como se muestra en la figura 1.6, incluso si se opta por varias particiones fijas pero de tamaños diferentes. En el caso de esta figura, la tarea T1 no cabe en ninguna partición, salvo que se permita mezclar particiones.

Por último, con este tipo de gestión aparece **fragmentación interna** debida al espacio que sobra dentro de cada partición al ubicar una tarea más

pequeña. Este efecto se puede observar en la figura 1.6, donde se comprueba que algunos recursos internos a las particiones ocupadas aunque están libres, ya no pueden aprovecharse.

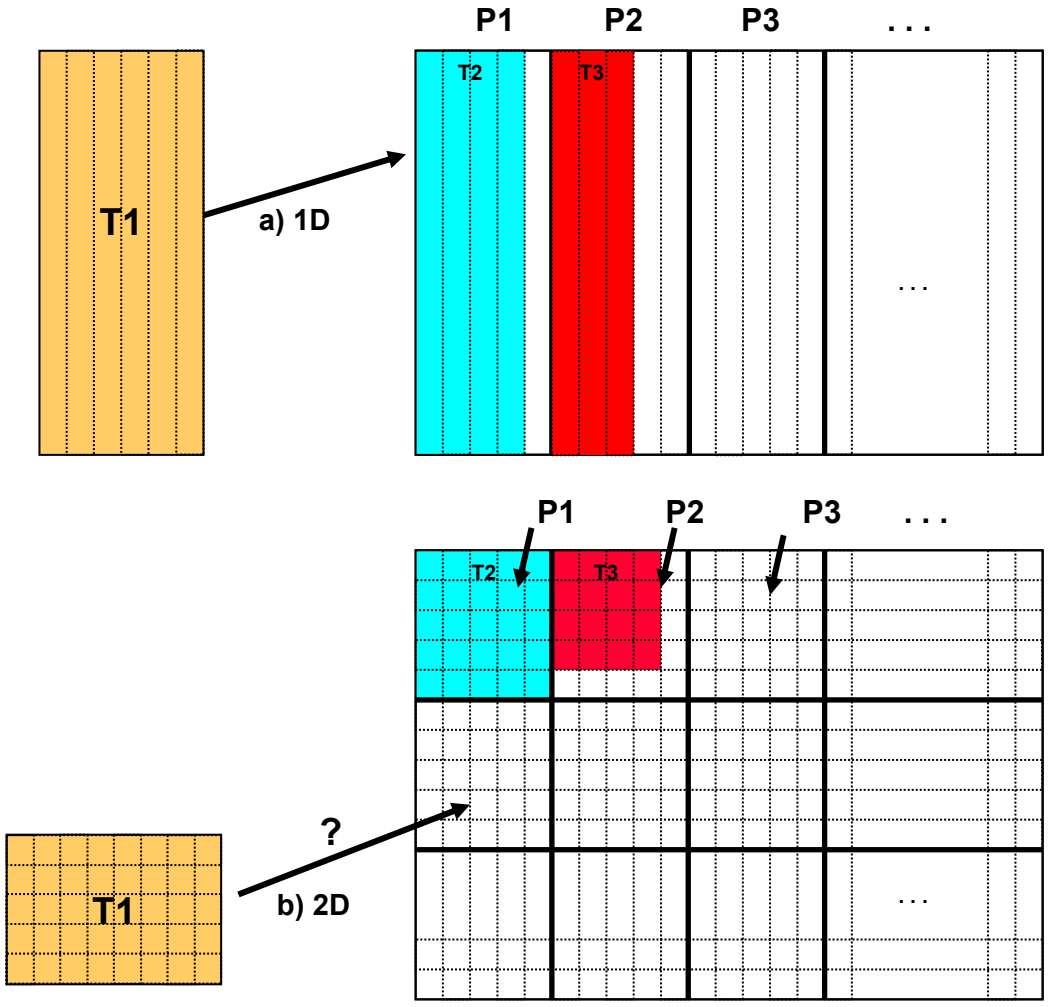


Figura 1.6. Gestión de particiones de tamaño fijo en a) 1D y b) 2D.

Si se considera la posibilidad de **particiones de tamaño arbitrario**, como se muestra en la figura 1.7, la gestión es análoga a la de la memoria segmentada paginada. Resulta más compleja, pero al mismo tiempo es más flexible. Las particiones pueden ajustarse al tamaño exacto de cada tarea, lo que permite un mejor aprovechamiento del espacio disponible. Cualquier BBR puede ser candidato a ubicar la tarea, pero los algoritmos de gestión del área libre son complejos y se debe intentar reducir el tiempo de búsqueda del espacio libre para ubicación de la tarea.

En este tipo de gestión, aparece **fragmentación externa** cuando salen de la FPGA tareas que han terminado su ejecución y dejan huecos libres difíciles de utilizar para ubicar otras tareas, tal como se muestra en la figura 1.7. Esto resulta especialmente difícil en un modelo de dos dimensiones ya que los huecos generados pueden tener formas totalmente arbitrarias.

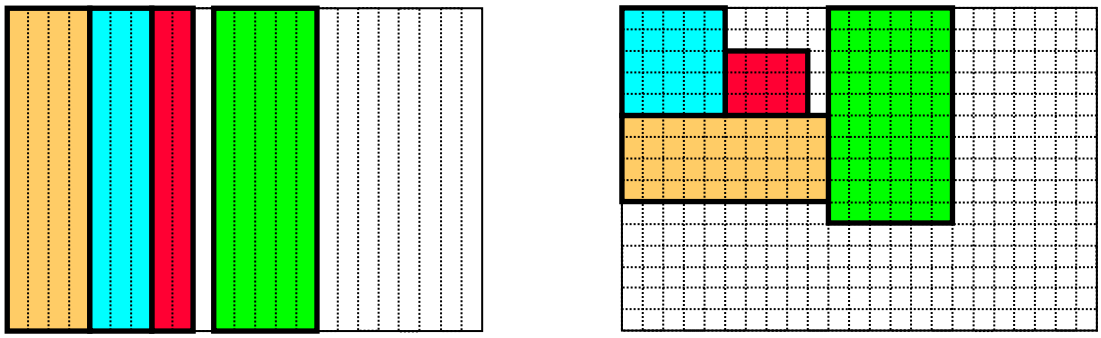


Figura 1.7. Gestión de particiones de tamaño arbitrario en a) 1D y b) 2D.

En cuanto a la **forma de las tareas**, la mayoría de trabajos previos limitan el análisis a la utilización de una única forma por tarea, imponiendo además la restricción de que dichas geometrías deban corresponder a rectángulos, con una anchura y una altura expresadas en número de BBRs [Dies98]. En algunos casos estos rectángulos pueden rotarse 90º para buscar una mejor ubicación, aunque desde el punto de vista de la manipulación de los mapas de bits esto parece puramente especulativo. Como alternativa, algún trabajo propone manejar diferentes geometrías o "formas" para una misma tarea HW, ya que algunas pueden resultar más adecuadas que otras para el mejor aprovechamiento del HW disponible (o incluso para que sea posible ese aprovechamiento). Incluso se especula con la posibilidad de tener formas totalmente arbitrarias para las tareas, como en [CLCK02] o [WaPl02], aunque al final las descartan y trabajan con formas rectangulares, ya que el uso de formas arbitrarias complica extraordinariamente la gestión. Como caso especial, se podría llegar a contemplar una alternativa en la que se pueda disponer de formas que ocupan más área pero son capaces de ejecutar la misma funcionalidad en menos tiempo, gracias a un grado mayor de

paralelismo, aunque esto complicaría extraordinariamente el espacio de diseño.

La figura 1.8 muestra varias formas alternativas de una misma tarea T_N, todas con el mismo área. Debido a la forma del espacio libre, sólo algunas de estas formas encontrarían una posición viable donde ser asignadas, mostrando la importancia que puede tener la selección de una forma HW concreta.

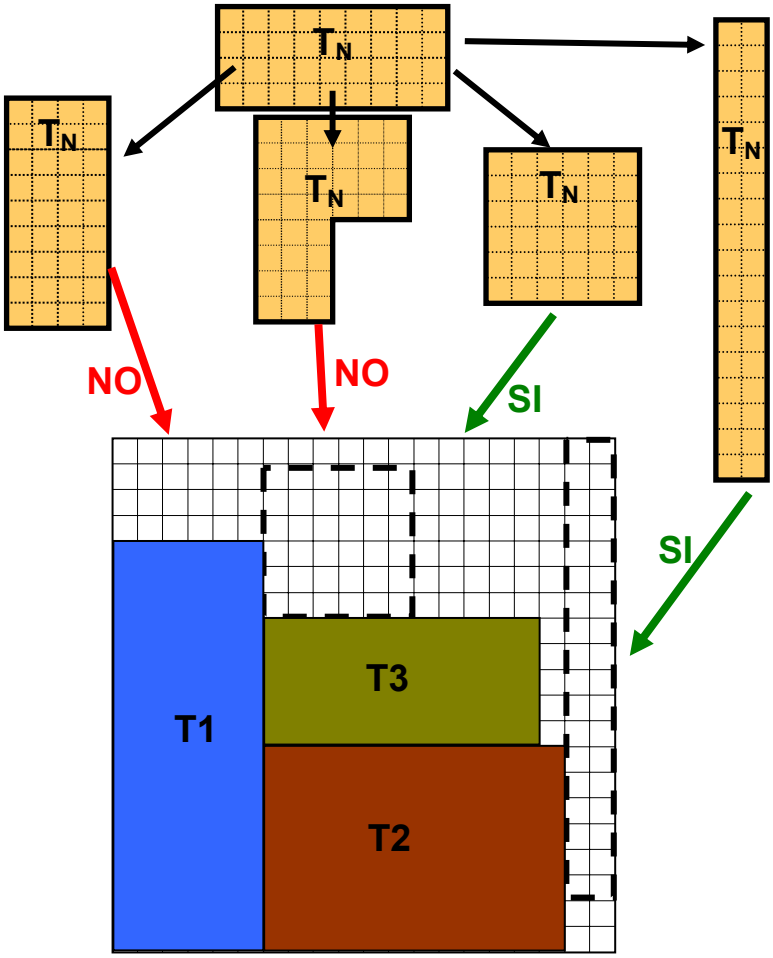


Figura 1.8. Formas de tarea alternativas.

e. Aceleración de la carga del código HW reubicable. La carga del código HW es un proceso que puede suponer importantes retardos porque conlleva la transferencia del mapa de bits desde la memoria en la que se almacena, hasta el HW reconfigurable (incluyendo la traducción de código reubicable a código absoluto, a partir de las decisiones de asignación de

HW), y la consiguiente reconfiguración dinámica de parte del mismo. En cierto modo, esta transferencia memoria/HW reconfigurable viene a ser el equivalente del cuello de botella procesador/memoria de la arquitectura von Neumann.

Por lo tanto, la minimización del tiempo de carga debe ser un objetivo básico, y se puede abordar bien planteando alternativas arquitectónicas, bien diseñando diversas técnicas de aceleración del tiempo de carga o bien mediante técnicas que minimicen el impacto de la reconfiguración en el rendimiento del sistema. Entre estas técnicas, se encuentra la inclusión de estrategias de **precarga de tareas HW**. Estas estrategias han sido abordadas desde la perspectiva de la prebúsqueda de partes de la aplicación, en entornos de codiseño HW/SW de sistemas empotrados sobre HW dinámicamente reconfigurable, como en [Hauc98], [NoBa01] y [RCGM08].

Las alternativas que existen son varias. En caso de disponer de un único contexto, la disponibilidad de una memoria rápida y de gran capacidad en el mismo chip que el HW reconfigurable se haría imprescindible. Esta memoria podría utilizarse para precargar las tareas HW planificadas, de manera que la carga pudiera hacerse luego de manera inmediata.

Otro enfoque propone reducir los tiempos de carga mediante técnicas de **compresión de las configuraciones**. Otro propósito que se consigue indirectamente al comprimir las configuraciones es el de reducir el tamaño de la memoria donde se almacenan las configuraciones, que se ha incrementado en los últimos años en la misma proporción que el tamaño de los dispositivos como se muestra en la sección 1.2. Las diferentes propuestas dentro de este enfoque varían, dependiendo de dónde se implemente el compresor y descompresor (externo o interno en la FPGA) y del rango de validez de la propuesta: si es válido para todo tipo de FPGA, o está desarrollado en concreto y adaptado a las particularidades de la arquitectura de una familia concreta.

La eficiencia de los algoritmos de compresión depende del tipo de interfaz de configuración y del formato del mapa de bits. A su vez, el formato del mapa de bits depende de las características del sistema de configuración y

de la arquitectura de la FPGA. Como consecuencia de lo anterior, el formato del mapa de bits varía entre fabricantes, e incluso entre las diferentes familias de FPGAs del mismo fabricante. En [DaPr01] y [DaPr05] se propone usar algoritmos de compresión basados en diccionarios, como el LZW [NeGa95], valido para cualquier FPGA porque procesa los mapas de bits como si fueran *raw data* (datos sin procesar). En [HUWB04] se propone el uso del algoritmo LZSS, (más fácil y rápido de implementar que el LZW) e incluye el descompresor dentro de la FPGA, conectando la salida directamente al interfaz SelecMap de una Virtex de Xilinx.

En la actualidad esta posibilidad de compresión de la configuración se incorpora en algunas FPGAs, como las últimas familias de Virtex. Esta opción es muy útil en algunos casos en los que se dispone de un diseño que cuenta con varios bloques de recursos HW idénticos, dispuestos de un modo muy repetitivo en una ruta de datos, ya que pueden configurarse varios frames de forma idéntica con una misma orden, independientemente de la utilización de alguna técnica de compresión de la configuración.

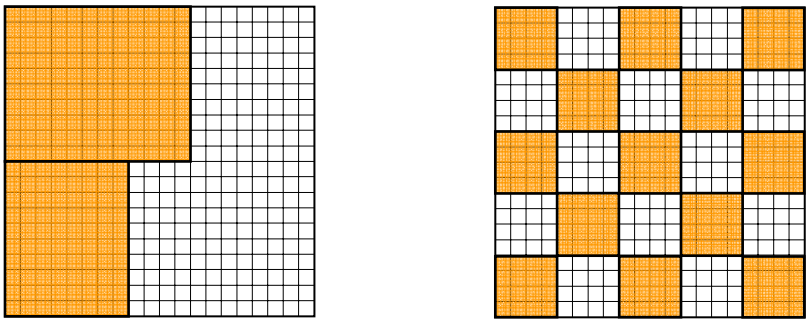
f. Estimación de la fragmentación. La fragmentación lleva a un rendimiento ineficiente del HWDR. Como ya se adelantó, pueden aparecer dos tipos de fragmentación: interna y externa.

La **fragmentación interna**, que fue descrita en la sección 1.3.d, es debida al tipo de gestión y organización realizada en la FPGA y no se pueden tomar medidas para corregirla. Como ya se expuso, cada tarea se puede modelar como una columna (1D) o un rectángulo (2D), sin embargo, la tarea únicamente utilizará un porcentaje de los recursos incluidos en dicha columna o rectángulo. El resto de recursos, que no se están utilizando de ninguna forma, no se pueden utilizar para ubicar otras tareas, dado que el SO considera que ya están asignados. Como ya se comentó anteriormente la definición del tamaño adecuado para las unidades básicas reconfigurables BBRs es un factor crítico para conseguir que se optimice el uso de los recursos. Si se elige un tamaño pequeño se evita que la fragmentación interna se dispare, pero los circuitos grandes deben dividirse en circuitos lo

suficientemente pequeños para que puedan implementarse en una unidad, y por tanto se complica el proceso de diseño y se puede reducir el rendimiento. Por otro lado, si el tamaño de los BBRs es muy grande, será posible implementar la mayor parte de los circuitos sin necesidad de dividirlos, pero en la mayor parte de los casos la fragmentación interna será muy elevada, y por tanto el porcentaje de recursos desaprovechados en cada unidad será muy alto. Si la cantidad de recursos es muy limitada, es imprescindible elegir un tamaño de BBR tal que la fragmentación interna no sea excesiva y que no sea necesario dividir de forma ineficiente los circuitos. Cuando el tamaño de las tareas es muy regular resulta fácil conseguir un tamaño óptimo, pero en caso contrario el tamaño que resulte más apropiado para las tareas más pequeñas no lo será para las más grandes. Para minimizar este problema, en [WaPl03] y [RMMS08] se propone utilizar unidades de distinto tamaño junto con un gestor de tareas que asignase cada tarea a la unidad cuyo tamaño es más conveniente.

Por otra parte, la **fragmentación externa** se produce por la finalización de tareas y entrada de otras nuevas en la FPGA y es susceptible de ser modificada. En esta situación existen recursos sin asignar a ninguna tarea distribuidos de tal forma que no resulta posible utilizarlos conjuntamente. Por ejemplo esta situación se puede dar si una determinada tarea ocupa *n* unidades de área y existen varios conjuntos disponibles, pero separados y de tamaño inferior. Aunque sumando el área de todos estos conjuntos el resultado sea mayor que *n*, no es posible utilizar estos recursos para ejecutar la tarea, a no ser que se realice un complejo proceso de interconexionado.

En la figura 1.9, se muestra una FPGA de 20*20 BBRs con un nivel de ocupación del 50%, pero con dos estados de fragmentación diferentes. En (a) se muestra una FPGA en un estado de baja fragmentación donde se podrían insertar tareas de hasta 8*20 BBRs, mientras que en (b) se muestra la misma FPGA en un estado de fragmentación mayor (puramente hipotético), donde no se podría insertar tareas mayores de 4*4 BBRs.



a) Baja fragmentación **b) Alta fragmentación**

Figura 1.9. Comparación de dos áreas fragmentadas.

En estas situaciones es muy útil utilizar criterios de ubicación de las tareas que tengan en cuenta los futuros estados de fragmentación, o bien realizar procesos de defragmentación cuando se alcancen unos niveles de fragmentación elevados.

Realizar la **estimación de la fragmentación externa** es imprescindible cuando la fragmentación externa crece y el área libre empieza a resultar poco aprovechable, para poder tomar medidas para corregirla y poder usar la FPGA de un modo óptimo. La fragmentación externa es más fácil de estimar y de combatir en una dimensión que en dos.

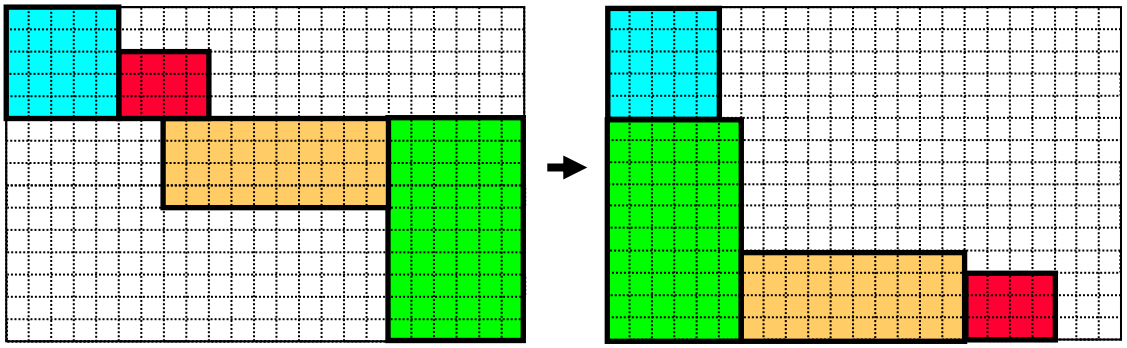
h. Defragmentación de HW mediante reubicación de las tareas ya en ejecución para permitir el reagrupamiento del área libre a fin de que pueda ser utilizado mejor. La defragmentación es una subtarea imprescindible, ya que es la única forma de garantizar el aprovechamiento óptimo del HW libre en cada momento.

La defragmentación podría realizarse bien a petición del propio gestor del HWDR cuando no se puede ubicar una tarea pero se detecta que hay recursos suficientes aunque fragmentados, o también de forma preventiva y automática cuando se den determinadas condiciones.

También puede abordarse con estrategias locales o globales, es decir, procediendo a defragmentaciones parciales, o a una defragmentación global

del HW. Estas estrategias deben intentar que la repercusión de la defragmentación sobre la ejecución de las distintas tareas en el HW sea mínima.

El problema de la fragmentación ha sido abordado por algunos investigadores proponiendo modificaciones en la arquitectura, para posibilitar la reubicación de una tarea sin necesidad de proceder a su recarga desde el exterior, aunque ello obliga a respetar como restricción el uso de la misma geometría (en el caso bidimensional) ya almacenada previamente. La figura 1.10 muestra un ejemplo de defragmentación, donde en (a) aparece una FPGA con un conjunto de tareas dispuesto de tal forma que el espacio libre está fragmentado, y después de realizar el proceso de defragmentación mediante la reubicación de algunas de ellas, el espacio resultante libre aparece unificado en un solo hueco de forma bastante uniforme, más capaz de albergar futuras tareas entrantes, tal como se muestra en (b).



a) Antes de la defragmentación **b) Después de la defragmentación**

Figura 1.10. Defragmentación del HWDR en dos dimensiones mediante reubicación de tareas.

i. Comunicación entre tareas reubicables. Se deben permitir las comunicaciones entre tareas de la misma o de diferentes aplicaciones y con el SO. Para realizar dicha comunicación se debe proporcionar un medio, que se puede implementar mediante un diseño de **buses** (en FPGAs con organización interna en 1D), y **redes** o **mallas** de interconexión (con

organización interna en 2D), que deben usar todas las tareas que se ejecuten en la FPGA. El problema de comunicación entre tareas siempre es más fácil de resolver para particiones fijas. Como ya se ha comentado en 1.3.d, será necesario que cada tarea incorpore un interfaz de E/S propio y específico para realizar sus comunicaciones. Las características de la E/S influyen por tanto en la elección del tamaño de BBR, porque la tarea mínima debe tener capacidad para incorporar dicho interfaz de E/S.

Una posible solución puede ser la utilización de un bus de E/S interno como el propuesto en [WaPl03b] para una arquitectura 1D, al cual tengan acceso todas las tareas, y que naturalmente habría que arbitrar para evitar conflictos.

En la figura 1.11, se muestra en (a) una FPGA con organización interna en una dimensión (columna) en la que aparece implementado un bus, para permitir realizar E/S así como comunicación entre tareas. En (b) se muestra una FPGA con una organización en 2D, donde aparece una malla de interconexión para permitir las comunicaciones.

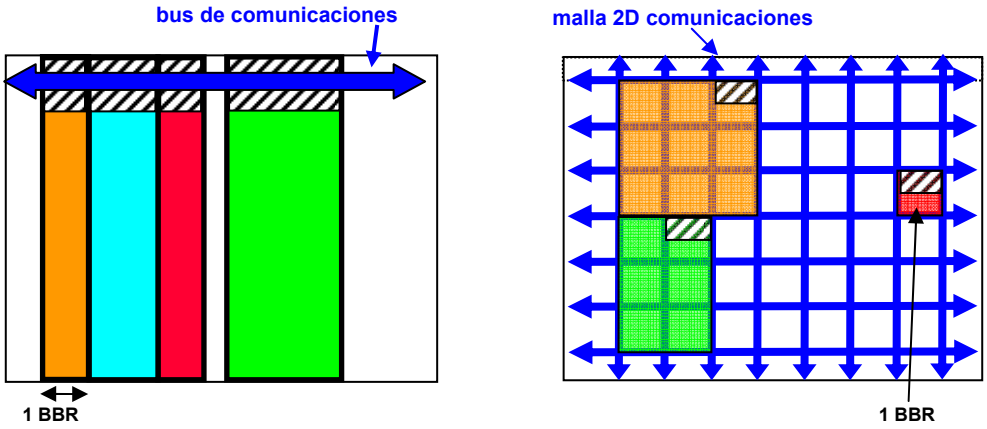


Figura 1.11. Sistema de comunicación 1D (a) y 2D (b).

En ambos casos, los sistemas de comunicaciones se deberían implementar de manera estática, reservando los recursos de interconexión disponibles en la FPGA y además las tareas deberían incluir un interfaz estándar de E/S para acceder al bus o a la malla de interconexión, tal como se representa en la figura 1.11.

Dentro de las FPGAs con organización interna en 2D, en las que se debe implementar una malla de comunicaciones, se puede optar por una solución basada en redes de comunicación On-chip, conocidas como networks-on-chip (NoCs), como la propuesta en [BAMT05]. Esta solución consiste en una malla 2-D de *routers* que realizan tareas de encaminamiento y direccionamiento simples. Este tipo de comunicaciones tienen la ventaja de permitir una gran variedad de patrones de comunicación, mientras que evita un proceso de diseño excesivamente complejo. Como desventaja se puede comentar la necesidad de volver a reconstruir los recursos de comunicación de la red (que se encuentran dispersos en todo el área de la FPGA) cada vez que una tarea finaliza su ejecución.

Otra implementación práctica sobre una FPGA de Xilinx de este tipo de redes de comunicación basadas en conmutación de paquetes se puede ver en el modelo ICN *(InterConnection Network)* propuesto en [MBVL02] y desarrollado sobre una FPGA Virtex, donde se divide la FPGA en un conjunto de unidades reconfigurables idénticas, en cada una de las cuales puede ejecutarse una subtarea. Cada una de estas unidades incluye un interfaz de comunicación fijo que permite la comunicación entre unidades usando primitivas de paso de mensajes sobre una red de interconexión que se encuentra integrada dentro de la FPGA.

Como reflexión final, conviene señalar que sería muy interesante que buena parte de las funciones y estrategias que se han ido proponiendo fuesen implementadas en el propio HW reconfigurable, posiblemente incorporadas a su arquitectura interna. Esto facilitaría la gestión eficiente de estos recursos (incluidos los de grano grueso) por parte del SO.

1.4 Objetivos de este trabajo

La propuesta de tesis doctoral que se plantea en esta memoria se centra en la **gestión de HW dinámicamente reconfigurable** de grano fino con un alto grado de homogeneidad en la distribución de recursos. Trata de resolver varios de los problemas más importantes planteados en este contexto, como

son la **gestión de información** sobre los recursos HW disponibles en cada momento**,** la **asignación** de recursos a tareas**,** la estimación de la **fragmentación** y la adopción de medidas de **defragmentación** del HW.

El sistema propuesto en este trabajo de investigación mantiene la información del área libre disponible con una estructura de datos basada en **Listas de Vértices (LV)**, que representa el perímetro del área ocupada. Por tanto, dicha lista de vértices es una representación geométrica del área disponible. En esta LV se consideraran sólo algunos vértices como posiciones candidatas a ubicar las nuevas tareas que van llegando a la FPGA, de modo que el proceso de selección de la búsqueda de ubicación de tarea se agiliza. El algoritmo utilizado para seleccionar la ubicación de la tarea elije el mejor vértice de acuerdo a una heurística dada. Los resultados experimentales mostrarán que debido a que se reduce de manera considerable el número de candidatos, el rendimiento, medido con unos parámetros objetivos que serán explicados en los capítulos 5 y 6, es superior a otras soluciones.

En este trabajo de investigación se ha implementado un entorno que simula un sistema de gestión de HWDR, que procesa un flujo de tareas de formas rectangulares y dimensiones arbitrarias en modo *online*, a medida que van llegando, y donde el dispositivo reconfigurable está organizado en dos dimensiones, con particionamiento arbitrario y arquitectura de grano fino homogénea. Sobre este entorno se ha implementado y probado una solución a la gestión de la información de recursos HW basado en la Lista de Vértices, y se han propuesto varias soluciones alternativas para la ubicación de tareas utilizando diversas heurísticas basadas en conceptos de adyacencia espacial, espacio-temporal y fragmentación. Para el problema de la estimación de la fragmentación se han desarrollado métricas que además de poder servir como heurística para realizar la ubicación de tareas, pueden usarse como alarmas para disparar distintas técnicas de defragmentación en situaciones con un alto grado de fragmentación. También se proponen técnicas para tratar este problema de la defragmentación del espacio libre.

La organización de la presente Memoria será como sigue:

En el capítulo 2 se revisará el trabajo previo relacionado con la problemática general que se trata en este trabajo de investigación: asignación de ubicación para las tareas, estructuras para mantenimiento del área libre, métricas de fragmentación, y técnicas de defragmentación.

En el capítulo 3 se presenta el núcleo del sistema de gestión de HWDR propuesto. Además se realiza una exposición detallada de la Lista de Vértices, que es la estructura utilizada para realizar la descripción del área libre del dispositivo.

En el capítulo 4 se presentan las métricas propuestas para medir el nivel de fragmentación del área libre y se realiza una comparación con otras alternativas planteadas por otros grupos de trabajo.

En el capítulo 5 se presenta un conjunto de heurísticas para la elección del vértice concreto donde se ubicarán las tareas, basadas en parámetros espaciales (2D), así como una comparación entre todas las propuestas.

En el capítulo 6 se presentan otras heurísticas más refinadas para la elección del vértice que combinan parámetros basados en el área y en el tiempo (3D) junto con resultados experimentales obtenidos en distintos tipos de escenario.

En el capítulo 7 se presentan las técnicas de defragmentación ideadas para tratar el problema de la fragmentación del espacio libre.

A continuación se plantean las líneas de evolución futura de este trabajo, y las conclusiones más relevantes.

Finalmente se incluye una bibliografía con las publicaciones a que ha dado lugar este trabajo y la bibliografía referenciada a lo largo de esta memoria.

Capítulo 2: Trabajo relacionado

Los principales problemas de gestión de recursos reconfigurables 2D, tales como **asignación** (de unos recursos concretos para la ejecución de una tarea), **planificación** y **fragmentación** del área libre, han sido tratados a lo largo de los últimos años por varios equipos de investigación. En este capítulo se revisarán estos problemas así como las soluciones propuestas por dichos grupos.

2.1 Gestión del área libre y ubicación de tareas

La asignación de recursos HW para ubicar una tarea HW dentro de una FPGA 2D es un problema semejante al clásico problema de

empaquetamiento de contenedores (*bin-packing*), que consiste en 2D, en colocar rectángulos arbitrarios dentro de un contenedor también rectangular. Este problema teórico ha sido estudiado en profundidad y se pueden encontrar soluciones genéricas en [CoCW02] con algoritmos First Fit (FF) y Best Fit (BF).

A continuación se muestra un resumen de los principales trabajos que han planteado soluciones (tanto teóricas como, en algunos casos, también prácticas) a la problemática expuesta.

2.1.1 Trabajo de O. Diessel

O. Diessel [DiEl01] parte de un modelo de FPGA en dos dimensiones con X*Y BBRs y con una estructura homogénea.

La organización interna de la FPGA es de particiones de tamaño variable y realiza ubicaciones de tareas en posiciones arbitrarias. El modelo de tarea propuesto tiene forma rectangular de tamaño l_x*l_y. El sistema tiene una estructura como la mostrada en la figura 2.1, en el que las tareas van llegando a través de una cola de tareas.

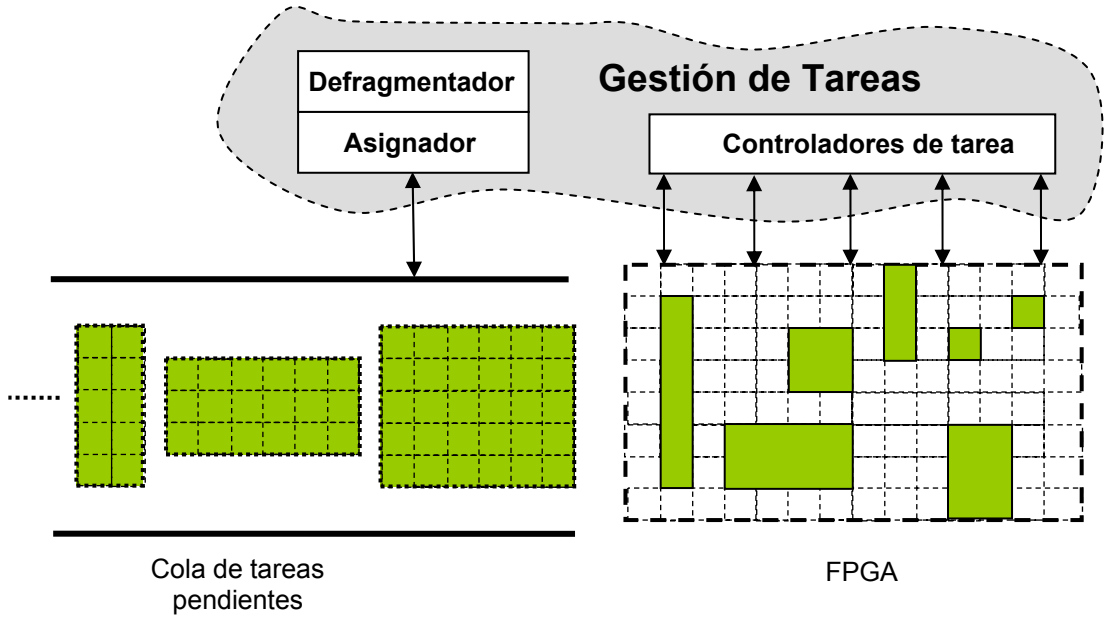


Figura 2.1. Estructura de Gestor de HW.

Diessel ha desarrollado una estructura en árbol para almacenar la información del área libre de la FPGA, que utiliza para asignar una zona libre de la FPGA a una tarea dada. Este árbol organiza el espacio libre de la FPGA de un modo jerárquico, particionando el espacio libre sucesivamente en cada nivel, en cuatro rectángulos de igual área que se marcan como completamente ocupado, completamente libre o parcialmente ocupado.

La representación del estado de la FPGA y su árbol correspondiente se puede ver en la figura 2.2. En la parte superior se muestra una FPGA con 16*8 BBRs y en la parte inferior la estructura de árbol donde se realiza la búsqueda. Para el estado actual de la FPGA con dos tareas en ejecución, no se podría ubicar una tarea de tamaño mayor de 4*4 BBRs porque aunque hay espacio disponible, pertenece a cuadrantes distintos.

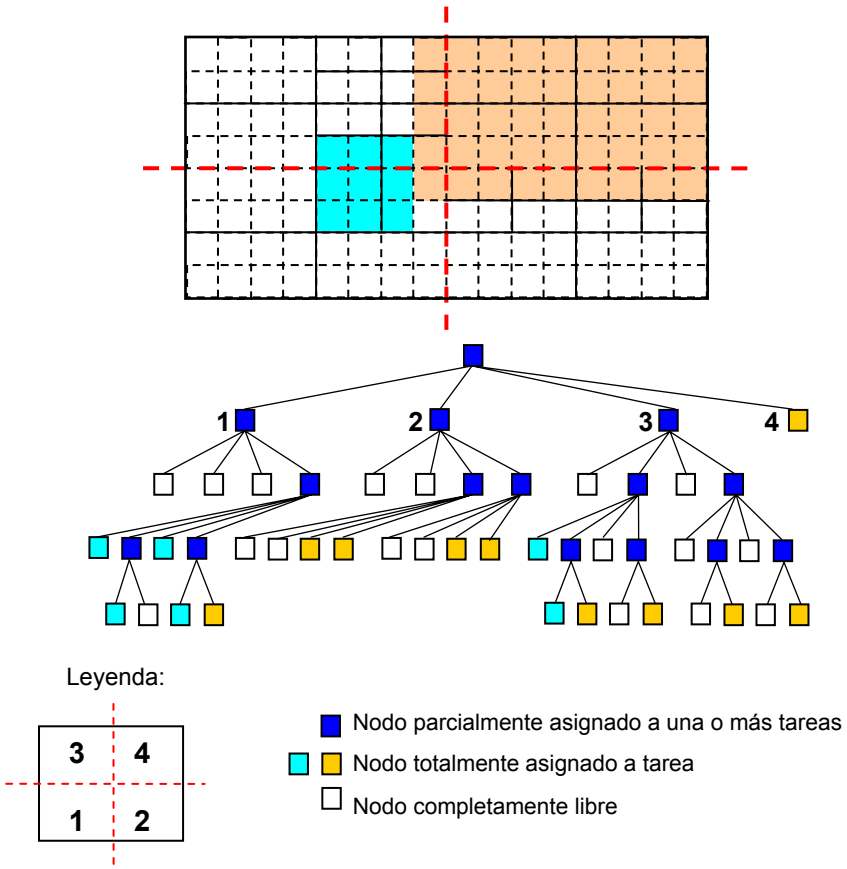


Figura 2.2. Representación del estado de FPGA con un árbol.

Esta estructura puede ser recorrida y actualizada muy rápidamente, pero no garantiza que se encuentre un sitio adecuado para una tarea entrante, incluso

cuando hay sitio suficiente, porque el espacio libre contiguo puede estar particionado a través de diferentes ramas del árbol. Además, esta solución no tiene en cuenta la fragmentación del área libre resultante cuando se selecciona la posición donde se ubica la tarea.

2.1.2 Trabajo de K. Bazargan

K. Bazargan [BaKS00] trata el mismo problema de ubicación de tarea usando soluciones basadas en *bin-packing* y aplicando algunos de los algoritmos clásicos para dicho problema teórico. Para ello propone diferentes estrategias para un *bin-packing* bidimensional-2D de tareas entrantes rectangulares. Estas estrategias difieren principalmente en el modo de gestionar el área libre. Dado que es uno de los trabajos de mayor relevancia y que sirve de referencia a muchos otros trabajos posteriores, será explicado con mayor nivel de detalle.

Bazargan propone un modelo de FPGA en dos dimensiones, homogénea y un Sistema mixto HW/SW, compuesto por una CPU más una RFU (Unidad reconfigurable). El sistema detecta si una tarea invocada está ya en el chip, de modo que no se necesita cargar de nuevo su configuración, y por el contrario, si la tarea fue previamente eliminada, el sistema ubica la tarea como si fuese la primera vez que se invocara. Cuando no hay sitio para ubicar una nueva tarea, el sistema la rechaza y se ejecuta en SW, incurriendo en una penalización en el tiempo de ejecución. En la figura 2.3 se muestra el modelo propuesto de FPGA organizada en 2D, donde hay algunas tareas en ejecución.

También se propone un modelo de tarea HW con dos variedades:

- Tareas rectangulares “a priori”: *“hard templates”.*
- Tareas organizadas jerárquicamente: *“firm templates”.*

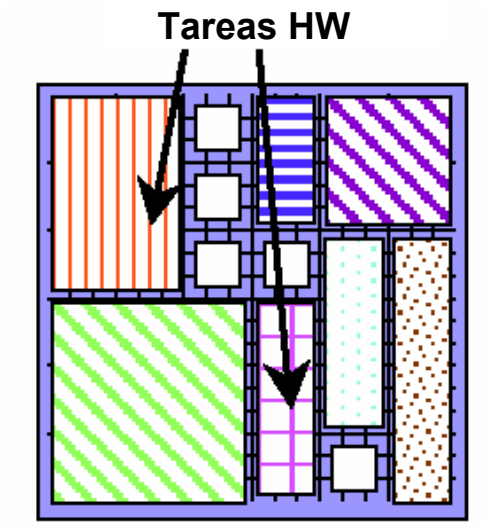


Figura 2.3. FPGA con operaciones en ejecución.

En el modelo "*firm templates*" las tareas están formadas por varias subtareas que pueden reorganizarse para facilitar la ubicación de la tarea global. En la práctica sólo considera (en sus algoritmos) las de tipo "*hard*" rectangulares. Las tareas se ubican en posiciones completamente arbitrarias, en particiones de tamaño variable.

Propone distintos algoritmos de asignación en dos contextos diferentes: asignaciones *online* (las tareas se procesan a medida que van llegando) y *offline* (se conoce de antemano el conjunto de tareas completo y se dispone de tiempo para encontrar una solución óptima).

En la asignación *online* (*bin-packing 2D*), donde las tareas se deben procesar a medida que van llegando, interesa la velocidad de procesamiento y se utilizan por tanto, algoritmos sencillos (FF, BF, etc.). Para poder cumplir con esos requisitos se necesitan estructuras de información de acceso rápido. Se proponen varias técnicas en las que el mantenimiento de la información del área libre se realiza sobre rectángulos solapados, o bien sobre rectángulos no solapados.

Para el contexto de asignación *online*, se propone el algoritmo KAMER (*Keep All Maximum Empy Rectangles)*. Para el mantenimiento de la información del área libre, se almacenan todos los rectángulos máximos

vacíos (MER, de *Maximum Empty Rectangles*) en una lista. Los MER pueden solaparse entre sí permitiendo la búsqueda de espacio libre para tareas de mayor tamaño. Cuando llega una tarea, se examina la lista de MER. Si hay un hueco en la FPGA, está necesariamente en algún MER. Si no hay hueco, se rechaza la tarea y en ningún caso se plantea realizar un proceso de defragmentación.

La elección del MER donde se ubicará la tarea se realiza entre los posibles que cumplen la condición ($A_{MER} > A_{tarea}$), mediante distintos tipos de búsqueda:

- First Fit (FF): el primero encontrado en el que cabe.
- Best Fit (BF): el que mejor se ajusta en tamaño (el más pequeño).
- Worst Fit (WF): el que peor se ajusta en tamaño (el más grande).

Otro proceso que se debe tener en cuenta es la actualización de la lista de MER. Al asignar un MER a una tarea, a partir del espacio sobrante se generan nuevos MER. Este proceso de actualización afecta a todos los otros MER con los que se solapa la tarea. Al terminar una tarea también hay que actualizar la lista. La complejidad de mantenimiento de la lista es elevada ya que el número de MER es de O(n*n), con n número de tareas.

La figura 2.4 muestra una FPGA con dos tareas en ejecución y los MER que representan el espacio libre. La ubicación de la nueva tarea dentro del MER elegido se realiza con la heurística **BL** (*Bottom-Left*, de esquina inferior-izquierda). En (a) aparece el estado inicial de la FPGA, donde el espacio libre está definido con 4 MER (etiquetados en la esquina inferior derecha como A, B, C y D) y además se indican los puntos donde se podría insertar la nueva tarea, V1 si se inserta en los MER A ó B, y V2 si se inserta en C ó D. En este ejemplo, el algoritmo de ubicación insertaría la nueva tarea en el rectángulo A que es en el único que cabe. En (b) se muestra el estado después de insertar la tarea con la actualización de los MER afectados que en este caso son A y B, que pasan a ser A´, B´ y E´. Por último, en (c) se muestra el estado después que una tarea en ejecución haya finalizado.

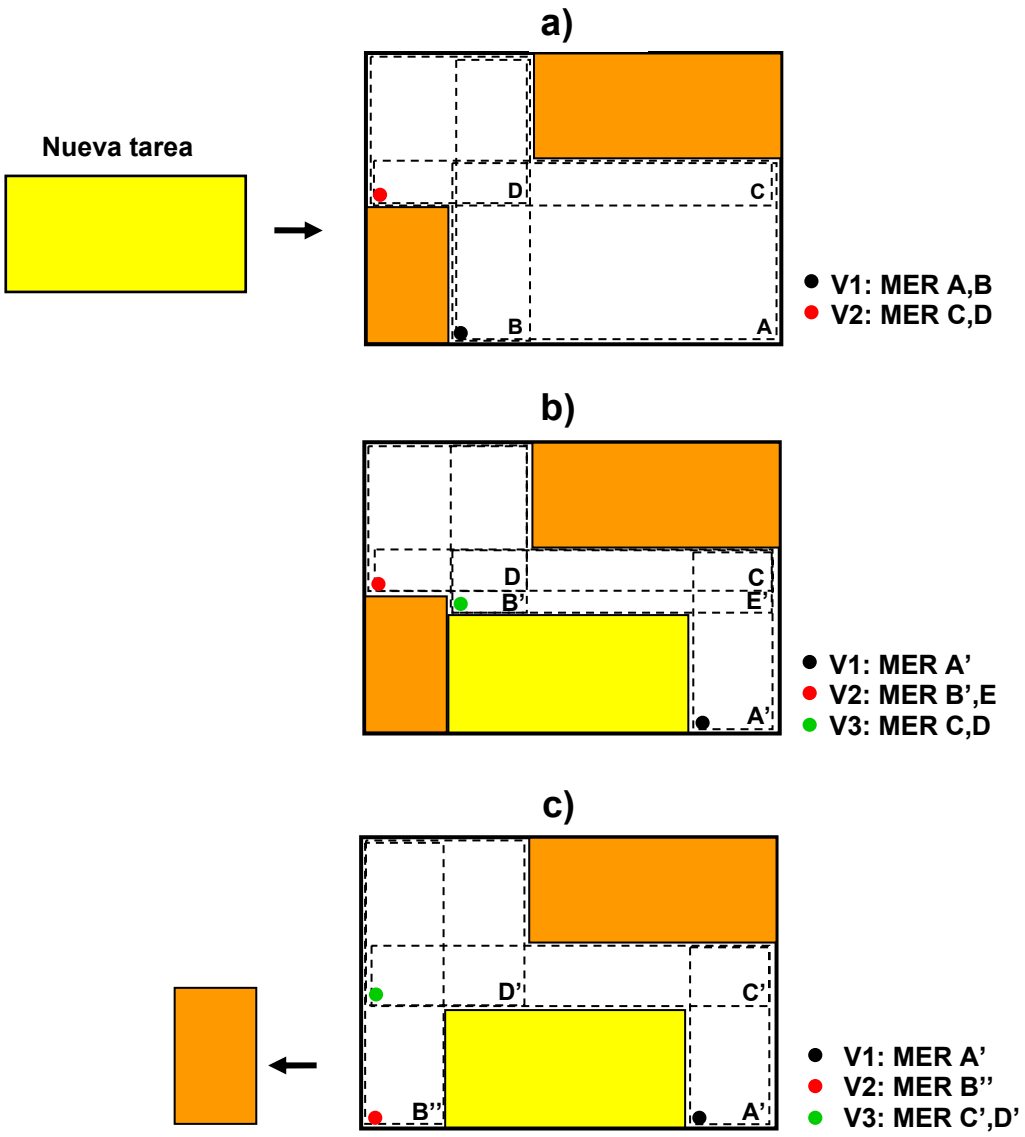


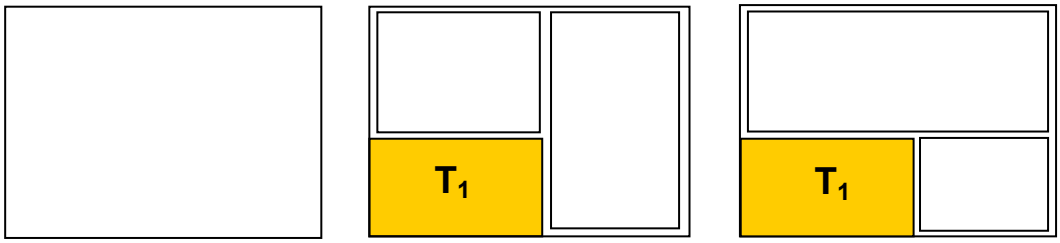
Figura 2.4. Representación del estado de la FPGA con MERs.

Por tanto, el algoritmo KAMER presentado, aunque utiliza la estructura MER que contiene todos los huecos disponibles, tiene como mayor desventaja la complejidad elevada de mantenimiento de la lista de MER. Además, las heurísticas utilizadas para la selección del MER son arbitrarias y la heurística BL para elegir la ubicación dentro del MER seleccionado tampoco es óptima.

Un segundo enfoque utiliza heurísticas para reducir el número de rectángulos afectados cuando se actualiza la lista de rectángulos y por tanto su complejidad. Cuando se selecciona un rectángulo libre para almacenar

una nueva tarea, el área resultante se divide sólo en dos nuevos rectángulos, que no se solapan. Estos rectángulos no son necesariamente máximos (NO-MER) y por tanto se pierde algo de calidad en las inserciones de tarea para poder reducir el orden de complejidad de la generación de la lista de rectángulos.

La figura 2.5 muestra el funcionamiento del particionador de espacio libre propuesto por Bazargan. El asignador ubica T_1 en la esquina BL de un rectángulo libre. El espacio libre resultante se puede dividir en dos rectángulos menores verticalmente (b), o bien horizontalmente (c). Como un rectángulo libre después de una ubicación se divide en dos nuevos rectángulos menores, dichos rectángulos son almacenados en un árbol binario usado para representar el estado de la FPGA, donde los rectángulos que estén libres en la actualidad son las hojas del árbol.



a) Rectángulo libre b) División vertical c) División Horizontal

Figura 2.5. Decisiones en la división de rectángulos no máximos.

La figura 2.6 muestra el proceso de fusión del espacio libre (si se opta por la división horizontal) después de borrar una tarea, que consiste básicamente en volver al estado anterior antes de que la tarea fuese insertada. En (a) se muestra una FPGA con una tarea T_1 y el área libre definido por dos MER no solapados A y B. En (b) la tarea T_2 se ubica en el rectángulo A, y divide el espacio residual de A, en C y D. Después de borrar la tarea T_2 en (c), los rectángulos C y D se eliminan y el rectángulo A se marca libre de nuevo.

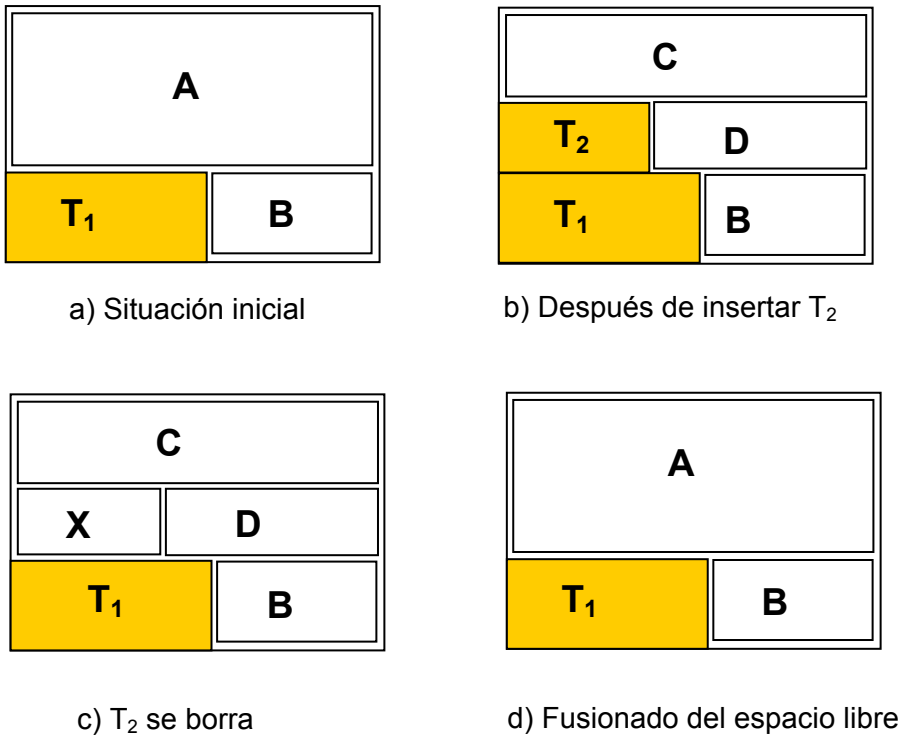


Figura 2.6. Ejemplo de fusión del espacio libre.

Un ejemplo de la gestión del área libre con rectángulos que no se solapan se muestra en la figura 2.7, donde en (a) se muestra el particionado realizado en el área libre con rectángulos no máximos (NO-MER1 a NO-MER4). Cuando llega una nueva tarea se tiene que ubicar en el rectángulo NO-MER3 porque es el único que cumple la condición ($A_{MER} > A_{tarea}$). En (b) se muestra el rectángulo NO-MER3 después de insertar la tarea y las opciones que existen para particionar el espacio resultante. Si se supone que se utiliza el segmento Sb (asumiendo que no existe Sa), cuando llega otra nueva tarea con unas dimensiones ligeramente inferiores al rectángulo (E,D), aunque podría ser asignado dentro del espacio libre, este método rechazaría la tarea porque no cabe ni en (A,B) ni en (C,D). Si el espacio libre se hubiese particionado usando el segmento *Sa*, la misma tarea sí podría ser insertada en el rectángulo (E,D). Por tanto el tipo de división elegido para formar los rectángulos no solapados no es trivial e influye en los resultados posteriores, perdiendo por tanto calidad en las ubicaciones de tareas.

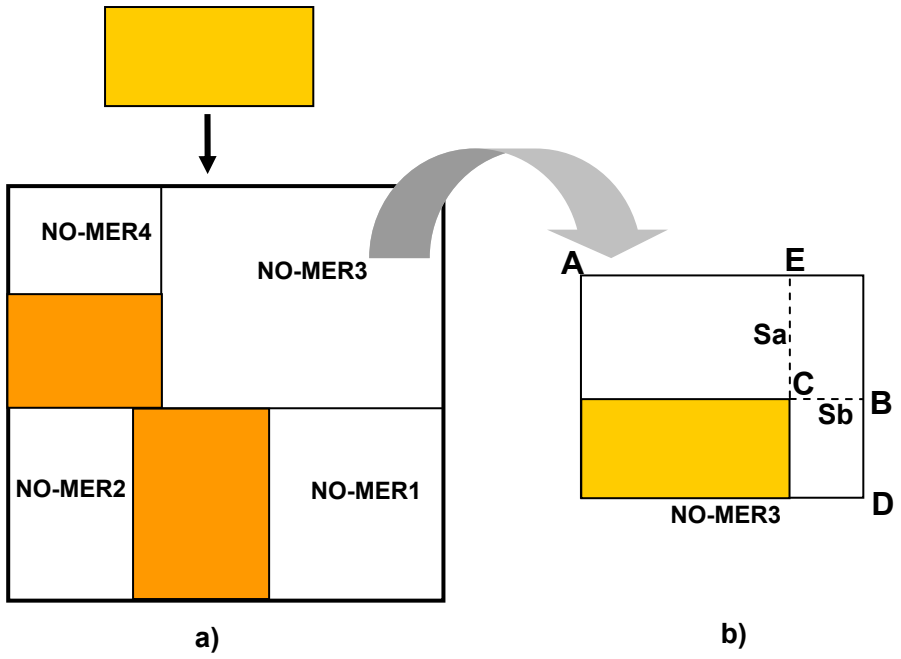


Figura 2.7. Particionado del área libre con rectángulos no máximos.

Utilizando este método se puede garantizar que el número de rectángulos considerados en la búsqueda es lineal en términos del número de tareas en la FPGA, con un orden de complejidad considerablemente menor que con los KAMER.

Para poder determinar el tipo de división que se realiza en el espacio libre se plantean varias heurísticas para la elección del segmento de división:

- Segmento más corto.
- Segmento más largo.
- Rectángulos más cuadrados.
- Rectángulo más grande más cuadrado.
- Rectángulos más desequilibrados en área:

  Mayor valor de $|W_1H_1 – W_2H_2|$
- Rectángulos más equilibrados:

  Menor valor de $|W_1H_1 – W_2H_2|$

La complejidad de este método es O(n*logn) para almacenar los rectángulos NO-MER y O(logn+K) para comprobar que los rectángulos vacíos pueden acomodar una nueva tarea, siendo k el número de posibles rectángulos candidato.

A pesar de que Barzagan presenta diferentes criterios para realizar la división en dos rectángulos, en sus resultados experimentales no se decide claramente por uno de ellos. De hecho no importa lo buena que sea la heurística, porque siempre hay posibilidades de realizar una división tal que lleve a que la siguiente tarea no se pueda insertar, incluso cuando existe espacio suficiente disponible. Tampoco aborda la problemática de la comunicación de tareas ni considera en modo alguno los recursos de E/S ni la fragmentación.

2.1.3 Trabajo de H. Walder

H. Walder propone en [WaSP03] una versión mejorada del particionador de espacio libre de Bazargan, con la misma complejidad pero con mayor calidad en las asignaciones de ubicaciones de tarea.

Utiliza modelos simplificados de tarea, en los que no hay precedencia ni restricciones temporales y además se impone la limitación de no poder reubicar la tarea una vez que se ha empezado la ejecución, con lo que se anula la posibilidad de realizar posibles procesos de defragmentación. En cuanto al modelo de FPGA en este trabajo, no trata E/S ni comunicaciones entre tareas.

La calidad de las diferentes asignaciones se estima mediante el tiempo de ejecución total y el tiempo de espera medio de un lote de tareas dado. Para tareas que son de la misma aplicación se busca minimizar el tiempo total del conjunto, mientras que para tareas no relacionadas se busca minimizar el tiempo de espera medio.

Utiliza como referencia la aproximación de Bazargan de rectángulos no solapados (NO-MER) para la definición del área libre.

La mejora sobre el proceso de Bazargan consiste en retrasar la decisión básica de división vertical u horizontal. Cuando se inserta una tarea en un rectángulo se crean dos rectángulos hijos máximos solapados y la decisión de la partición no se toma hasta que no llega una tarea que vaya a ser asignada a A ó B. Si se inserta en A, entonces B se modifica para que no se solape con A, y viceversa, una ubicación de la tarea en B lleva a modificar el ancho de A.

El particionador propuesto, denominado *On-the-fly* (OTF), retrasa la decisión de partirlo incluso más. La figura 2.8 muestra un ejemplo que ilustra el funcionamiento. La tarea T_1 se ubica en una posición BL dentro de la FPGA (a) y se crean dos rectángulos hijos de acuerdo a la versión mejorada OTF. Cuando llega T_2 (b) se inserta en el rectángulo B y el particionador mejorado modifica entonces el rectángulo A. Sin embargo el rectángulo T_2 no se solapa con A, y por tanto se puede dejar el rectángulo A en su tamaño original, obteniendo un mejor particionado del espacio libre como se muestra en (c).

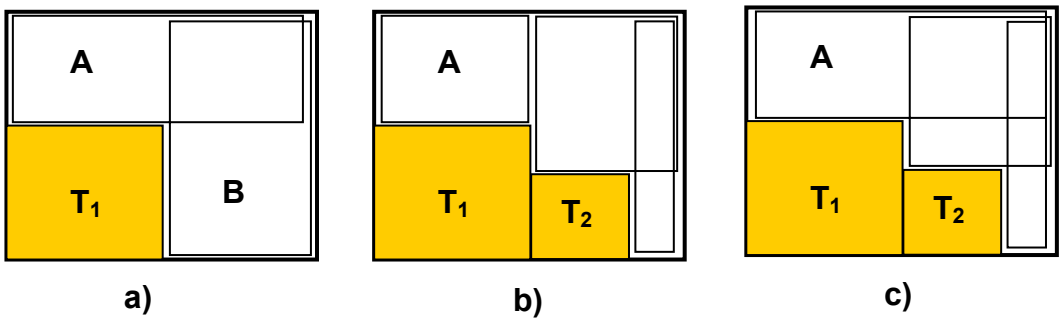


Figura 2.8. Rectángulos hijos solapados máximos.

El precio a pagar es que puede ser necesario modificar un mayor número de rectángulos después de insertar o borrar una tarea. La figura 2.9 muestra el proceso de actualización de rectángulos, extendiendo la situación mostrada en 2.8.c.

La figura 2.9.b muestra el resultado de insertar una tarea T_3 en el rectángulo A que se solapa con B. En este caso la ubicación de T_3 no afecta sólo al rectángulo A sino que también se necesita actualizar el subárbol completo de

rectángulos que sale de B (en este caso se tiene que modificar la altura de los rectángulos del subárbol). La implementación del particionador OTF se diferencia de la mejorada de Bazargan solamente en que todos los rectángulos del subárbol podrían ser modificados en un instante de tiempo posterior.

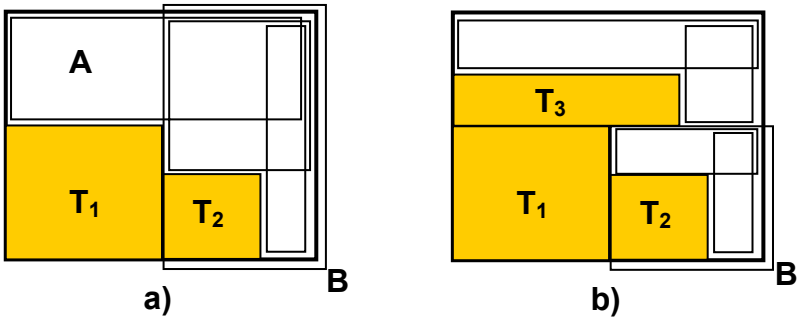


Figura 2.9. Modificación de rectángulos hijos.

Este grupo de investigación también presenta en [WaSP03] una propuesta de matriz *hash* para encontrar una posición en tiempo constante, pero la actualización de esta estructura tiene un alto coste temporal. En la solución de Bazargan los rectángulos están localizados en las hojas del árbol que representa la superficie de la FPGA. Cuando llega una nueva tarea el Gestor de HW tiene que buscar en la lista el rectángulo libre adecuado según la heurística seleccionada. El proceso de actualización requiere operar en el árbol binario.

El nuevo enfoque de Walder da prioridad a la búsqueda rápida para asignar recursos a una tarea cuanto antes. Para acelerar la búsqueda se mantiene una matriz *hash* en paralelo al árbol binario. Además se permite que el proceso de actualización de la matriz *hash* se realice en paralelo a la ejecución de la tarea. Este compromiso puede penalizar en exceso en entornos de trabajo en los que la latencia de llegada de tareas sea pequeña, o incluso cuando puedan llegar varias a la vez.

Dada una FPGA de tamaño X*Y, se define la matriz *hash* como un array *ar* de tamaño X*Y elementos. Un rectángulo libre de tamaño a*b está asociado con la entrada *ar[a,b]*. Cada entrada consiste en un puntero a una lista de

rectángulos libres del tamaño correspondiente y denominado puntero libre. La figura 2.10 muestra una matriz *hash* de X*Y elementos, donde se detalla el contenido de la entrada *ar[a,b]* que contiene el puntero de la lista de rectángulos libres.

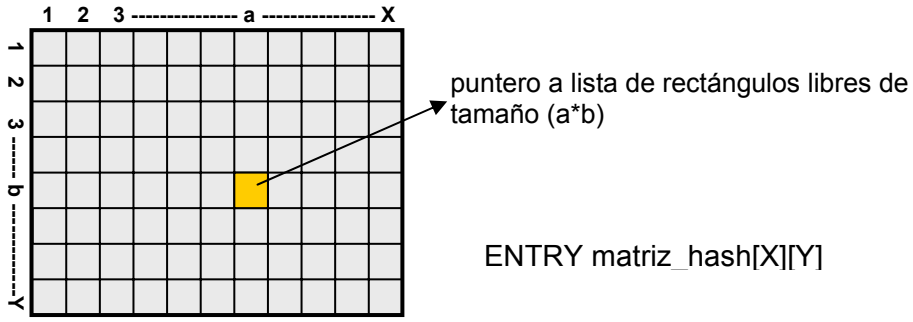


Figura 2.10. Ejemplo de Matriz hash.

Siempre que se ubica o se borra una tarea, los punteros libres de algunas entradas tienen que actualizarse. La figura 2.11 muestra este proceso. Inicialmente en (a), la matriz *hash* mantiene un rectángulo vacío R1, reflejando que toda la superficie de la FPGA está vacía. Si se inserta una tarea de 3*5, se borra el rectángulo vacío y se insertan dos nuevos rectángulos libres R2 y R3 en la matriz *hash*. Durante este proceso, todos los punteros libres tienen que actualizarse, de modo que las entradas que pertenecen a rectángulos con tamaños hasta 5*7 apuntan a R2 y todos los rectángulos con altura 1≤Y≤2 y anchura entre 6 y 8 apuntan a R3. Todas las demás entradas apuntan a *null* como se muestra en (b). Después de que finalice la tarea, R2 y R3 se borran y se inserta de nuevo R1. En este punto la situación inicial se restablece, como se muestra en (c).

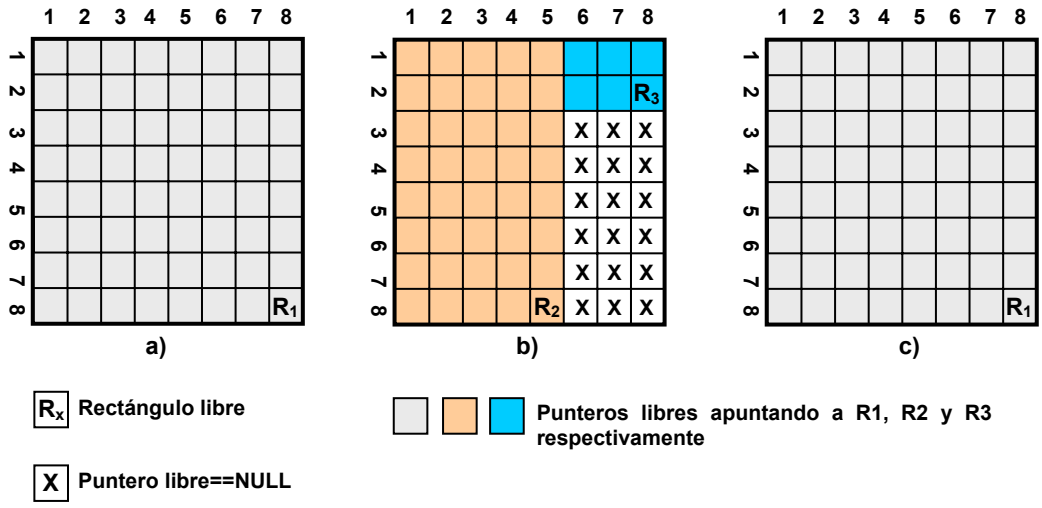


Figura 2.11. Actualización de punteros libres en matriz hash.

Además de esta mejora a la técnica de Bazargan, Walder propone en [StWP04] un sistema para gestión de HWDR que realiza ubicación de tareas *online*. Presenta dos heurísticas, las técnicas de *Horizon* y *Stuffing* para la ubicación de tareas y considera los recursos de la FPGA como modelos en 1D y 2D. En ambos modelos se reserva una parte del dispositivo reconfigurable para funciones del SO. El resto de área es para la ejecución de tareas HW.

Los resultados experimentales muestran mejor rendimiento en las heurísticas de *Stuffing* a costa de una mayor complejidad y que además los resultados obtenidos para 1D dependen mucho de la proporción del aspecto de las tareas, según sean más alargadas o cuadradas.

En el modelo 1D más simple, las tareas pueden ser ubicadas en cualquier posición a lo largo de la dimensión horizontal del dispositivo; la dimensión vertical está fijada y se expande a la altura total del área del hardware. El modelo de área en 1D, más realista y adaptado a la tecnología actual disponible en ese momento, lleva a problemas de ubicación y planificación simplificados. Sin embargo, este modelo sufre de dos tipos de fragmentación. El primero, de modo interno, se produce por el desperdicio de área cuando una tarea no utiliza la altura total del área del dispositivo. El segundo, de naturaleza externa, aparece cuando el área libre queda dividido en varias

columnas no conectadas. Dicha fragmentación puede impedir que una tarea pueda ser asignada, aunque haya suficiente espacio libre disponible. En este trabajo no se han tomado medidas para abordar este problema.

El otro modelo 2D permite asignar tareas a posiciones arbitrarias en el área de tareas HW y sufre menos fragmentación interna. Consecuentemente, se obtienen mejores niveles de utilización, pero a costa de una planificación y asignaciones más complejas.

2.1.4 Trabajo de A. Ahmadinia

A. Ahmadinia [ABBT04] propone un sistema de reconfiguración dinámica que se muestra en la figura 2.12, que consta de una CPU, memoria externa donde se almacenan las tareas y un dispositivo reconfigurable. El Planificador gestiona las tareas y decide cuándo y en qué dispositivo (hardware o software) se debe ejecutar la tarea.

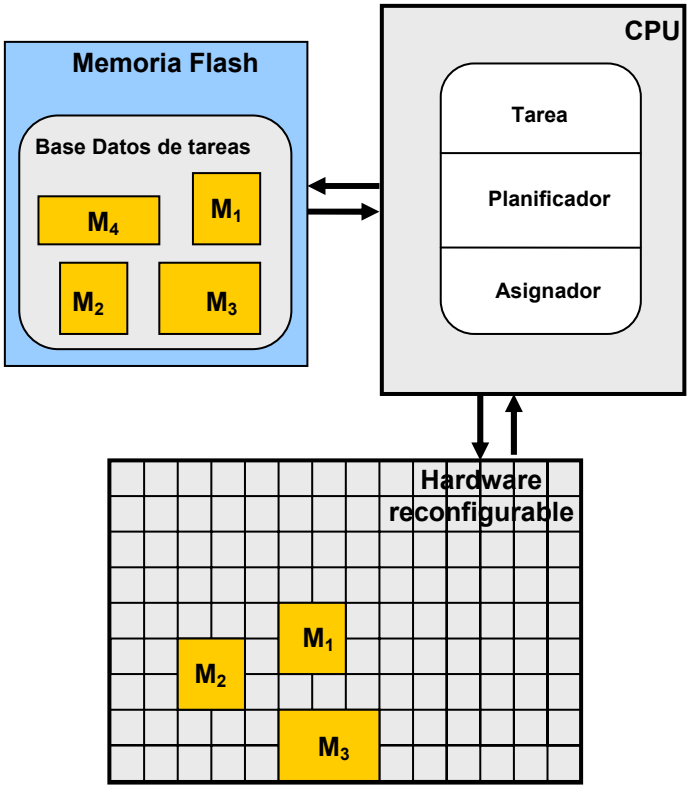


Figura 2.12. Sistema multitarea en un dispositivo reconfigurable.

Para tareas que se van a ejecutar en hardware el planificador realiza una petición al asignador de ubicación de tarea, que trata de buscar espacio libre para que la tarea se pueda ejecutar. Las tareas son rectangulares y están definidas por la tupla de parámetros similar a la utilizada en este trabajo, pero en la que no hay restricciones temporales para su ejecución.

Para la gestión del área del dispositivo reconfigurable presentan otra versión mejorada del método de asignación de ubicación de tarea de Bazargan, donde se gestiona el área ocupado en lugar del área libre, de este modo, en la mayoría de los casos el número de rectángulos es mucho menor, aunque el orden de complejidad del algoritmo es el mismo (n*logn), con *n* el número de tareas. La heurística usada para asignar una tarea intenta además minimizar la distancia a la tarea o tareas con la que comunica. Las tareas se asignan en posiciones BL y se define la región de colocación imposible (IPR, de *Imposible Placement Region*) para evitar solapamiento entre tareas.

La figura 2.13, muestra un ejemplo de cómo se calcula la región IPR cuando en el dispositivo hay una tarea *c´* en ejecución y llega un nuevo componente o tarea *c*, con altura h_c y ancho w_c. La región de colocación imposible IPR denominada Ic´(c) de *c* respecto de *c´*, se identifica calculando el margen izquierdo con tamaño w_c*-1* y el margen inferior con tamaño h_c *-1* de la tarea *c´* donde la nueva tarea no puede ser ubicada porque se solaparía con la tarea en ejecución *c´*.

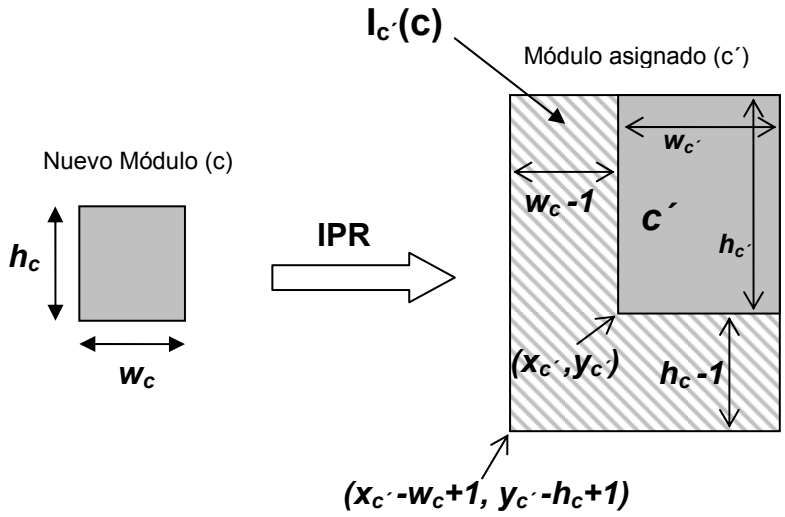


Figura 2.13. IPR de una tarea nueva *c*, relativo a una tarea en ejecución *c´*.

Cuando se realiza para todas las tareas que hay en ejecución, este cálculo se debe extender y hay que considerar dichos márgenes para todos los módulos. Si calculamos la IPR relativo a todas las tareas en ejecución y al dispositivo como en el ejemplo anterior, se obtiene un conjunto de IPR como se muestra en la figura 2.14. Aparecen detalladas la región IPR Ic´(c) de *c* respecto de la tarea *c´* y la región $I_R(c)$ de *c* respecto del dispositivo reconfigurable *R*, calculadas como en la figura anterior. De igual modo se calculan las regiones IPR para todo el conjunto de tareas que hay en ejecución y la nueva tarea c no podrá ser ubicada en ninguna de las zonas rayadas que aparecen en la figura 2.14. La región donde es posible asignar la tarea *c* (PPT, de *Possible Placement Regions*), se obtiene finalmente restando el conjunto de IPRs de la superficie total del dispositivo.

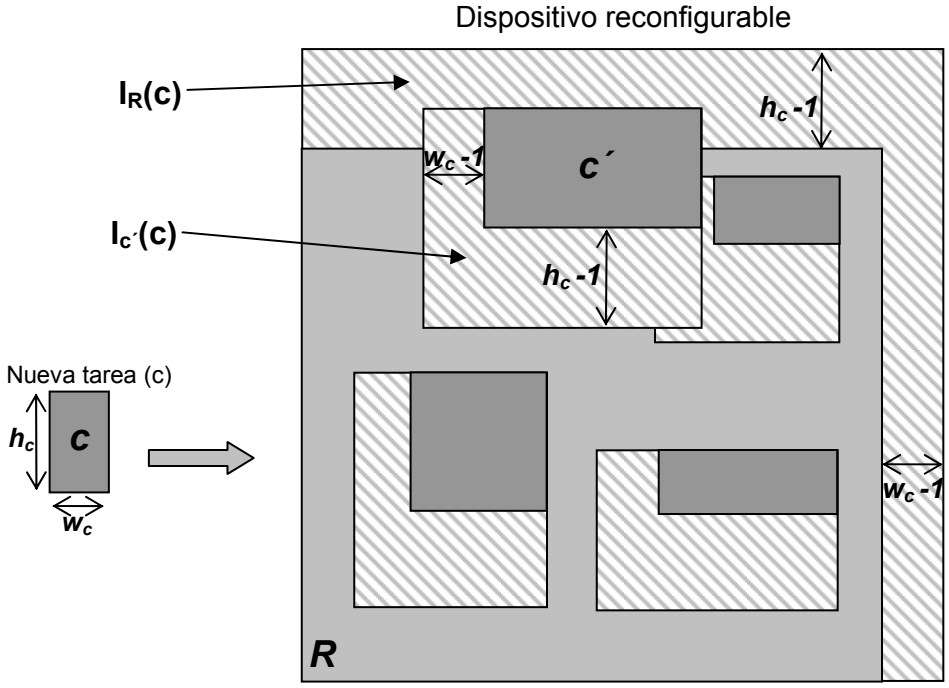


Figura 2.14. IPR del dispositivo relativo a todas las tareas en ejecución.

El segundo subproblema relacionado con la asignación *online* es encontrar la posición óptima para ubicar la nueva tarea de todo el conjunto de regiones posibles PPT. En lugar de calcular el coste de la asignación para cada posible ubicación, se calcula directamente el punto óptimo *popt* que proporciona el valor máximo de la función de coste de ubicación. Si *popt* se

encuentra dentro de PPR, entonces se dispone de una solución al problema. En caso contrario, se busca el punto más cercano a *popt* que esté localizado dentro de la región PPR y se selecciona como punto óptimo para la asignación. La función de coste de ubicación intenta optimizar las comunicaciones entre tareas, situando tareas que tienen conexiones comunes lo más cercanas posibles, e intentando minimizar el coste de comunicación de todo el conjunto de tareas en ejecución en el dispositivo, en términos de distancia y ancho de bus.

En la figura 2.15, se muestra en (a) una tarea en ejecución, el punto óptimo para insertar la nueva tarea que está dentro de la región IPR (zona rayada) y los puntos más cercanos posibles. En la parte inferior (b) se muestran estos puntos para un conjunto de cuatro tareas en ejecución.

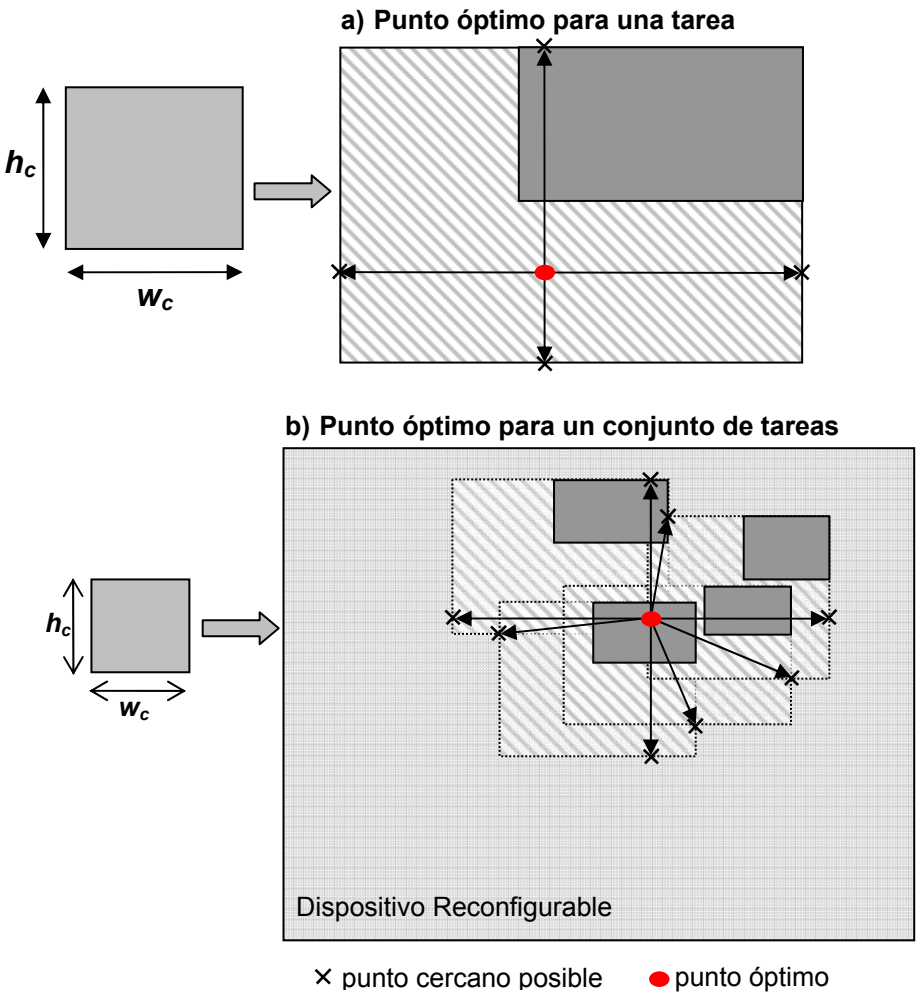


Figura 2.15. Punto óptimo de ubicación de una nueva tarea, para una (a) y para cuatro tareas en ejecución (b).

Para implementar el algoritmo de asignación *online* utilizan una lista de tareas en ejecución de tamaño O(n) y una matriz bidimensional de las mismas dimensiones que el dispositivo reconfigurable para representar el estado del dispositivo, de modo que cada elemento refleja el estado de cada unidad mínima reconfigurable. Además utilizan una matriz bidimensional dinámica para almacenar el coste de las comunicaciones entre cada par de tareas en ejecución. Esta matriz se utiliza para elegir la posición óptima para situar una nueva tarea.

Adicionalmente, Ahmadinia considera en [AhBT03] dos metodologías de agrupamiento (*clustering*) para ubicar tareas en tiempo real en dispositivos reconfigurables trantando de reducir el coste de la comunicación y utilizar los recursos de modo más eficiente. El primero de ellos se basa en agrupar tareas que tengan tiempos de ejecución similares, mientras que el segundo se dirige a reducir el coste de comunicación con estrategias que tienen en cuenta un compromiso entre comunicación entre tareas y área utilizado.

Como principal limitación puede señalarse que no tiene en cuenta la fragmentación generada en el área libre, ni las restricciones temporales o de datos durante la planificación de la tarea. Además, se utilizan demasiadas estructuras de datos muy complejas de actualizar.

2.1.5 Trabajo de M. Handa

M. Handa [HaVe04] propone un sistema para gestión de HWDR que realiza ubicación de tareas *online*. Cada aplicación se divide en varias tareas rectangulares, a las que se les asigna un tiempo de comienzo cuando hay espacio libre disponible en el dispositivo, pero en caso contrario no hay listas de espera o planificación, sino que son rechazadas y se ejecutan en software. La superficie de la FPGA se modela en un array bidimensional, con X columnas e Y filas, llamado Matriz de área. Cada celda del array representa un CLB de la FPGA.

La figura 2.16 muestra la matriz de área de una FPGA con dos tareas situadas en ella. Cada celda de la matriz de área tiene un peso asociado: si está vacía se representa por un número positivo y si está ocupada por un número negativo. Un número positivo en una celda proporciona el número de celdas contiguas libres situadas por encima, incluyéndola a ella, y un número negativo en una celda ocupada refleja el ancho remanente de la tarea medido hacia la derecha desde esa celda. En la figura se muestra en la columna de la derecha de la matriz, la celda P codificada con el número 9, que indica que todas las celdas de la columna están libres, mientras que en la misma columna, la celda M, indica que hay 5 celdas contiguas libres situadas encima. La estructura de datos de matriz de área se puede actualizar de manera eficiente después de añadir o borrar una tarea. Este método de codificación acelera el proceso de búsqueda si lo comparamos con otros en los que la codificación se realiza con 0´s y 1´s como en [EGLM03] y [AGSU87].

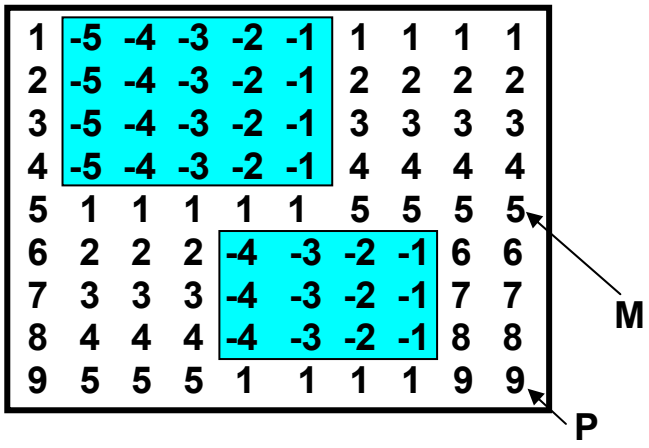


Figura 2.16. Matriz de área de una FPGA.

A partir de esta Matriz se presenta un algoritmo rápido para encontrar hueco, que utiliza una estructura de datos basada en escaleras. La figura 2.17 muestra algunas escaleras en una FPGA en la que hay varias tareas en ejecución. La escalera en el punto P tiene tres escalones con aristas A, B y C. La escalera con origen en M tiene sólo un escalón con una arista C.

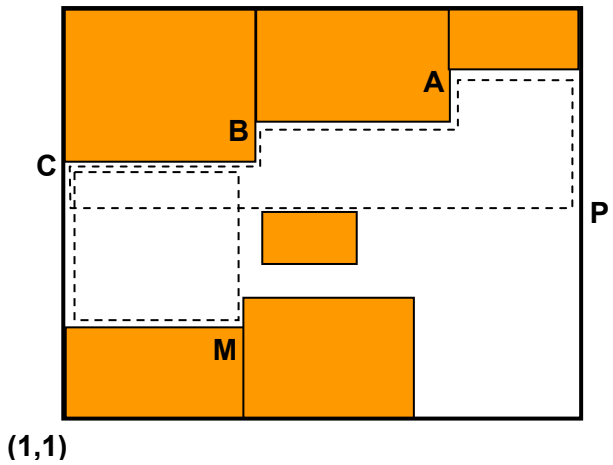


Figura 2.17. Ejemplo de escalera en una Matriz de área.

Para formar la escalera se escanea la *Matriz de área* fila a fila desde arriba hacia abajo y se hacen escaleras en las posiciones libres.

La figura 2.18 muestra el proceso de construcción de una escalera. En la primera entrada positiva más a la izquierda de cada fila escaneada, hay sólo un peldaño de la escalera cuya altura esta dada por el número positivo almacenado en esa posición de la *Matriz de área*. Una escalera en la posición (*x+1*, *y*) se puede construir fácilmente desde la escalera en el punto (*x,y*), mostrada en línea punteada. Siendo (*x*, *y')* el punto mas arriba y a la derecha del peldaño que contiene la columna *x*, mientras (*x+1*, *y''*) es la coordenada de la celda vacía que este más arriba en la columna *x+1*. En la figura se muestran los tres posibles casos:

- Si *y''* > *y'* , se añade un nuevo peldaño a la escalera y el punto de mayor altura de la escalera es *y''*.
- Si *y''* = *y'*, el ancho del peldaño más a la izquierda se extiende uno.
- Si *y''*< *y'*, todos los peldaños con altura mayor que la del punto *y''* se borran y el ultimo paso borrado se reemplaza con una altura igual a *y''*.

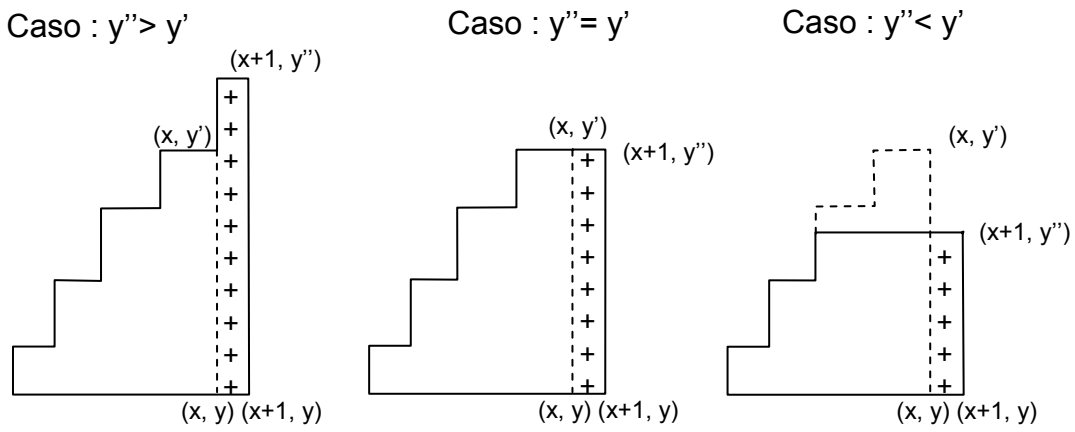


Figura 2.18. Ejemplo construcción de escalera.

A partir de esta estructura de datos en escalera, es posible extraer los MER teniendo en cuenta que algunas escaleras pueden no contener MERs. Para acelerar el proceso de identificación de MER, no se analizan todas las escaleras, solamente se comprueban aquellas que descansan en el perímetro derecho de la matriz de área o el peso de aquellas cuyo origen es mayor que el de la celda situada inmediatamente a su izquierda.

De modo similar a las soluciones propuestas en este trabajo de investigación, este algoritmo para encontrar rectángulos libres se puede utilizar para resolver distintos problemas como son la ubicación de tareas, la defragmentación o la reubicación de tareas. Los resultados experimentales muestran que sólo se necesita explorar cerca del 15% de las celdas de la FPGA para hacer escaleras, con lo que se reduce drásticamente el tiempo de búsqueda de espacio libre y por tanto el de ubicación de tarea. Sin embargo, la búsqueda en dicha estructura tiene una complejidad de O(X*Y), donde *X* e *Y* son los números de columnas y filas respectivamente. Además no hay detalles acerca de los criterios de selección de los MER, ni se considera la complejidad del algoritmo de ubicación de tareas.

2.1.6 Trabajo de H. Kalte

H. Kalte propone en [KKK04] un sistema implementado sobre una Virtex de Xilinx. El sistema propuesto se muestra en la figura 2.19, donde la FPGA se gestiona en 1D, apropiado para el sistema de configuración de la familia Virtex. Para la comunicación entre módulos y para realizar E/S, se propone una infraestructura de comunicación de tipo bus que permite no sólo comunicación entre módulos vecinos, sino también entre dos o más módulos situados en cualquier parte del dispositivo. Para su implementación se propone un bus triestado clásico, que se expande horizontalmente a lo ancho de todo el dispositivo. Este bus consta de segmentos múltiples de conductores y *buffers* triestado. En la figura se muestran dos segmentos de bus separados por un módulo *Bridge* que divide el bus horizontal en dos segmentos independientes. La función de los *Bridges*, entre otras, es bloquear las comunicaciones locales dentro de cada segmento y permitir las comunicaciones entre módulos a través de los distintos segmentos. Además se propone la utilización del protocolo AMBA bus para poder facilitar su integración en la mayoría de bloques prediseñados.

En el proceso de inicialización se envía un mapa de bits inicial para configurar el bus, la infraestructura de comunicación, la Unidad de gestión de memoria (MMU, de *Memory Management Unit*), el Árbitro y los módulos estáticos: Asignador de recursos, Gestor de configuraciones y Unidad de E/S. Posteriormente varios mapas de bits parciales especifican el comportamiento de los módulos que son cargados de manera dinámica. En la figura se muestran además varios módulos dinámicos como CPU, Decodificador, DSP y CRC.

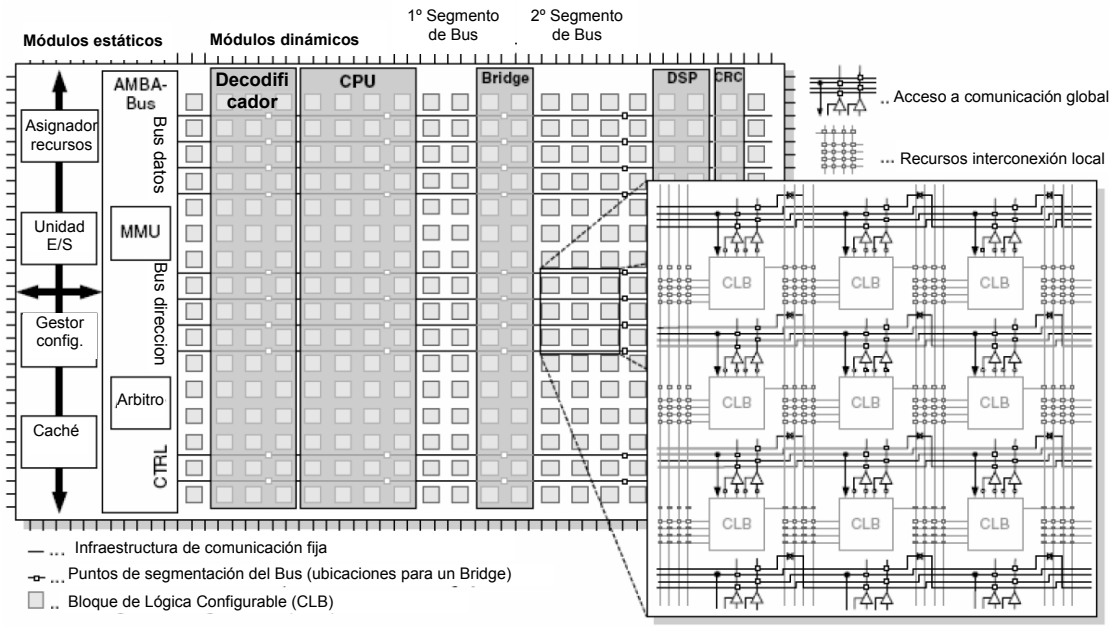


Figura 2.19. Entorno multitarea 1D sobre Virtex.

La reubicación en tiempo de ejecución de los módulos presintetizados a lo largo del bus horizontal, se basa en la manipulación del mapa de bits original, donde se cambia la dirección de columna de los módulos individuales. Antes de empezar su ejecución todos los módulos se sintetizan a un área específica de la FPGA, y dentro de ese área la E/S del módulo se conecta al bus triestado para proporcionar acceso al bus de comunicaciones durante su ejecución (como se muestra en la zona ampliada donde cada CLB se conecta a los hilos triestado disponibles en las Virtex). La principal restricción de la solución propuesta es que se utilizan todos los hilos triestado y no pueden ser utilizados para las interconexiones internas del módulo. Otra limitación supone que esta solución no está orientada a las familias modernas de Xilinx con posibilidad de reconfiguración real en 2D. Para optimizar el uso de la FPGA y minimizar la limitación que supone la reconfiguración 1D, se propone compactar los módulos a lo ancho, y utilizar toda la altura del dispositivo. De ese modo se reduce la fragmentación interna reduciendo el número de columnas necesarias al mínimo, sin efectos negativos en el rendimiento.

Finalmente los resultados experimentales sobre una Xilinx Virtex 2000, comparan su propuesta de sistema de gestión 1D, con otro sistema en 2D basado en el método KAMER de Bazargan (usando una heurística Best Fit), y

muestra que la utilización de recursos de la FPGA, medidos en CLBs es superior en el 1D respecto del 2D. Sin embargo, el banco de pruebas está limitado ya que se utiliza sólo cinco tamaños de módulo posibles, con tres relaciones de aspecto fijas (1:1, 1:2, 2:1) y con unos tiempos de ejecución fijos proporcionales al tamaño de los módulos.

En [KoPK06] se presenta un algoritmo de ubicación de tareas para FPGAs 1D heterogéneas, donde se considera la existencia de diferentes tipos de recursos como procesadores empotrados, memoria o unidades aritméticas, denominados componentes estáticos. En la práctica, para simplificar y sin que se pierda generalidad, sólo considera como elementos estáticos los bloques de memoria. La idea es que las tareas HW que usan esos componentes estáticos (memoria) no pueden estar situadas en cualquier posición del array reconfigurable, y sus posiciones viables están limitadas por las de los bloques de memoria. Sin embargo para las tareas básicas, que tienen muchas más posiciones viables (ya que no incluyen bloques de memoria), se intenta que no ocupen posiciones que puedan ser usadas por aquellas tareas que tienen pocas posiciones viables.

Para aplicar el algoritmo se define la probabilidad de utilización de cada una de las posiciones libres donde se puede alojar la tarea. Se basa en la idea de que una tarea sólo puede ser ubicada en posiciones donde aparezca el mismo patrón con los distintos tipos de celdas definidos para la tarea.

En la figura 2.20 se muestra en la parte derecha una FPGA con dos tipos de elementos distintos A y B, que se podrían corresponder a CLBs y BlockRAMs. En la parte izquierda aparecen los patrones de dos tareas m_1 y m_2, donde se especifica el tipo de celda y su posición. Esta restricción limita la posibilidad de utilización de cada una de las celdas. Para la tarea m_1 las posiciones viables serán sólo las 1 y 5, mientras que para m_2 serán 1, 4, 5, y 8.

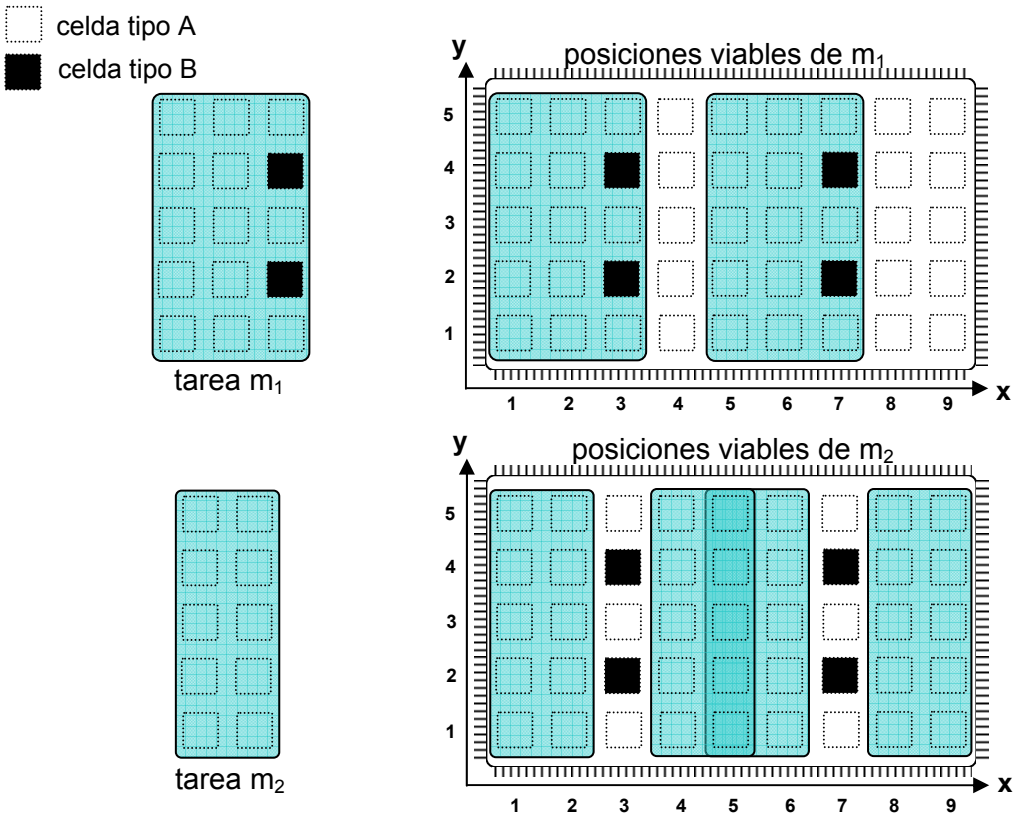


Figura 2.20. Patrones de tarea en FPGA heterogénea 1D.

Teniendo en cuenta las posiciones viables para cada tarea, se crea la Tabla de Cobertura de posiciones viables para las tareas m_1 y m_2 mostrada en la tabla 2.1.

Tabla 2.1. Tabla de Cobertura de posiciones viables.

x	1	2	3	4	5	6	7	8	9
$o_{pos}(m_1, x)$	1	1	1	0	1	1	1	0	0
$o_{pos}(m_2, x)$	1	1	0	1	2	1	0	1	1

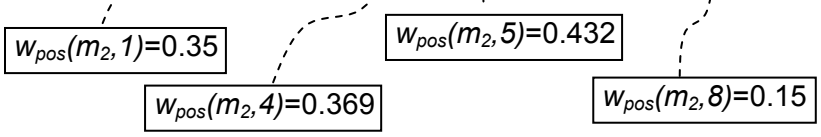
Además de la Tabla de Cobertura, la probabilidad de utilización de una columna también depende de la probabilidad de que cada tarea sea invocada por el sistema. Utilizando la tabla anterior y la probabilidad de utilización de

cada tarea, se calcula la probabilidad de utilización de cada posición $p_{pos}(x)$, que describe la probabilidad de que la columna (x) sea utilizada por una tarea HW.

La tabla 2.2 muestra la probabilidad de utilización $p_{pos}(x)$ para las tareas m_1 y m_2 cuando las probabilidades de invocación de cada tarea son 0,4 y 0,6 respectivamente. Teniendo en cuenta que la columna en $x=5$ puede ser usada por m_1 en la posición viable $x=5$ y por m_2 en las posiciones $x=4$ y $x=5$, tiene la mayor probabilidad de utilización $p_{pos}(5)=0,5$. Además para cada tarea m_i se calcula el peso de cada posición $w_{pos}(m_i, x)$, que es la media cuadrática de la probabilidad de utilización $p_{pos}(x)$ de todas las columnas que ocupa la tarea. En la tabla además aparecen los pesos de cada una de las posiciones viables para la tarea m_2.

Tabla 2.2. Probabilidad de utilización de cada posición.

x	1	2	3	4	5	6	7	8	9
$p_{pos}(x)$	0.35	0.35	0.2	0.15	0.5	0.35	0.2	0.15	0.15



Por tanto, el algoritmo de asignación 1D intentaría insertar la tarea m_2 en aquellas posiciones viables libres que tengan un menor peso (en este caso la 8), y dejaría libres las posiciones de mayor probabilidad de ocupación (la 5) para que las tareas con mayores restricciones, como m_1, puedan encontrar sus posiciones viables sin ocupar. Los pesos anteriores se calculan de manera estática antes de la ejecución, y las posiciones viables se ordenan de acuerdo a este peso, de modo que para la tarea m_2, el orden de búsqueda de una posición viable será 8, 1, 4, 5.

Este algoritmo aplicado a 2D aparece en [KoPK05]. En la figura 2.21 se muestran los patrones de dos tareas m_1 y m_2, donde las posiciones viables

para m_1 son (1,1), (5,1), (1,3) y (5,3) y las posiciones viables para m_2 son (1,1,), (4,1), (5,1) y (8,1).

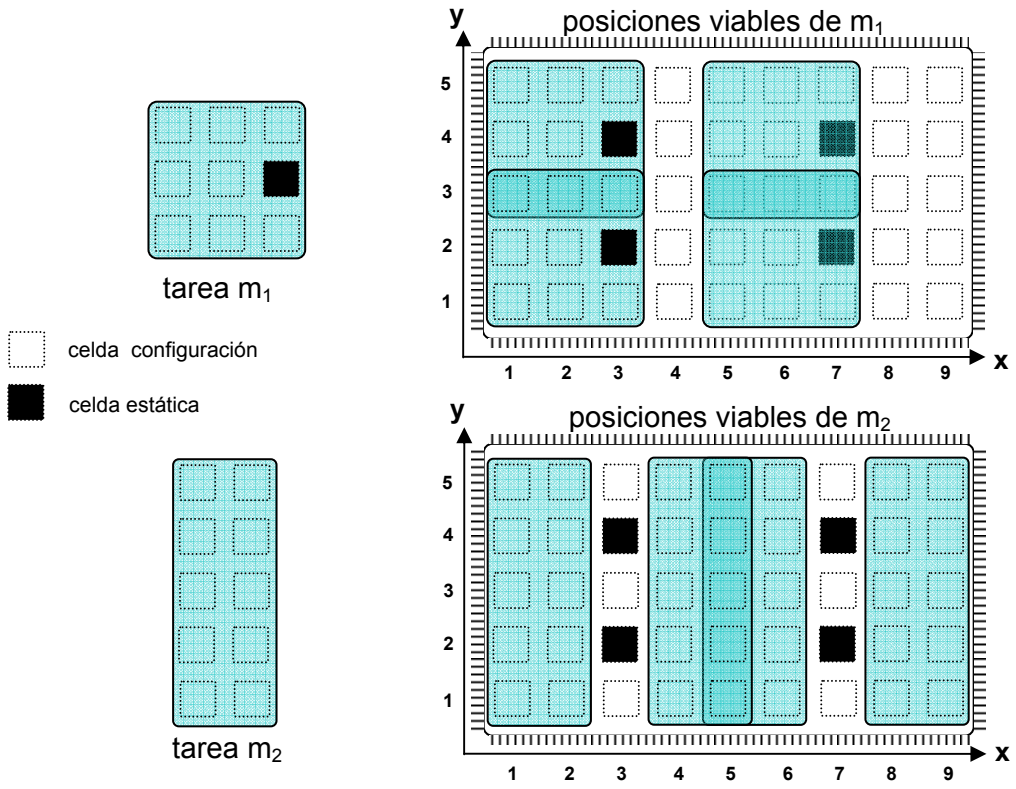


Figura 2.21. Patrones de tarea en FPGA heterogénea 2D.

De un modo similar al caso 1D, se calcula la tabla de Cobertura de posiciones viables y se calcula la probabilidad de utilización de cada posición, como se muestra en la figura 2.22. En la figura aparece la probabilidad de utilización calculada para cada celda de la FPGA y aparece el peso para el caso de la tarea m_2. También en este caso el algoritmo de asignación situaría la tarea m_2 en la posición viable libre que tenga menor peso, en este caso la (8,1), para dejar libres aquellas posiciones de mayor probabilidad de ocupación como la (5,1).

En este caso el cálculo de la probabilidad se realiza de modo estático con el algoritmo SUP Fit (*Static Utilization Probability*) y de manera dinámica con el algoritmo RUP Fit (*Run-Time Utilization Probability*). La principal diferencia radica en que para el RUP Fit, los pesos de las posiciones se actualizan cada vez que se asigna o se elimina una tarea. Además el RUP Fit considera dos

parámetros adicionales: la prioridad relativa de cada tarea (que se ajusta dinámicamente en tiempo de ejecución) y la frecuencia de utilización de cada tarea, de modo que el algoritmo evita ubicar una tarea que se carga raramente en posiciones viables de una tarea que se carga con mucha frecuencia.

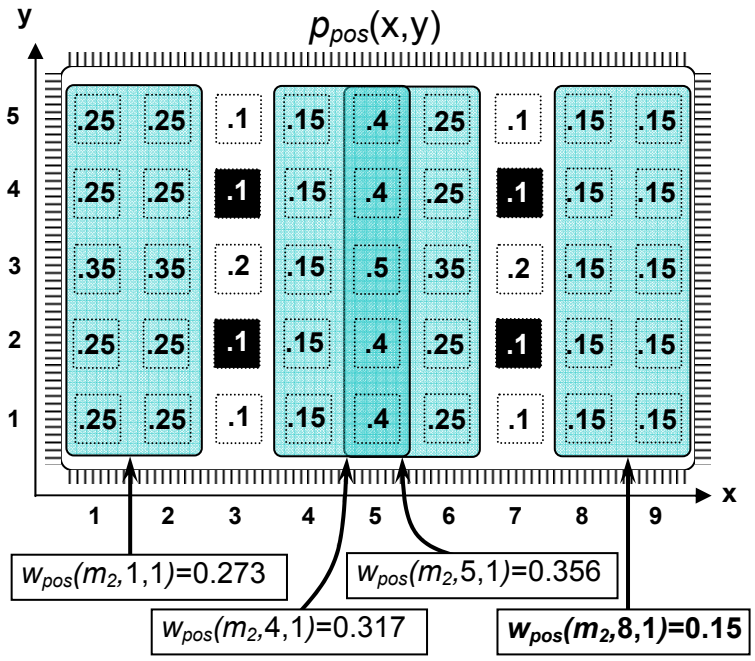


Figura 2.22. Probabilidad de utilización de cada posición para la tarea m_2.

Los resultados experimentales muestran que los algoritmos basados en la probabilidad de utilización de las posiciones viables (SUP y RUP Fit) mejoran el rendimiento cuando se comparan con el clásico BestFit. Sin embargo, aunque el RUP Fit mejora los resultados, es extremadamente complejo de calcular e introduce demasiado *overhead* en los tiempos de ejecución de las tareas. Tampoco tiene en cuenta la fragmentación del espacio libre que supone una solución frente a otra.

2.2 Fragmentación: métrica y técnicas de defragmentación

Dentro de la gestión de la ubicación de tareas en los recursos reconfigurables, uno de los problemas más importantes es cómo minimizar la fragmentación generada al realizar esta ubicación en tiempo de ejecución.

Este problema en 1D es más simple y es conceptualmente idéntico al de la fragmentación en las páginas de memoria virtual. Sin embargo en 2D es más complejo de estimar y de resolver.

Algunos grupos de investigación han propuesto métricas de fragmentación que se utilizan para ayudar a sus algoritmos de ubicación para elegir la posición más adecuada para las tareas entrantes. Otros grupos además realizan un proceso de defragmentación explícito.

2.2.1 Trabajo de K. Compton

K. Compton [CCKH00] considera la reubicación como respuesta al problema de fragmentación, aunque no presenta ninguna métrica de fragmentación. En este trabajo se propone una arquitectura 2D denominada R/D FPGA (RD, de *Relocation/Defragmentation*) que es capaz de reubicar tareas residentes (en ejecución) en un modo fila a fila, con la ayuda de un buffer adicional del tamaño de una fila para tener espacio suficiente para tareas que llegan.

Esta arquitectura R/D FPGA se muestra en la figura 2.23, en la que aparecen sus principales componentes como el área de volcado (buffer adicional), el decodificador de fila y algunos registros adicionales.

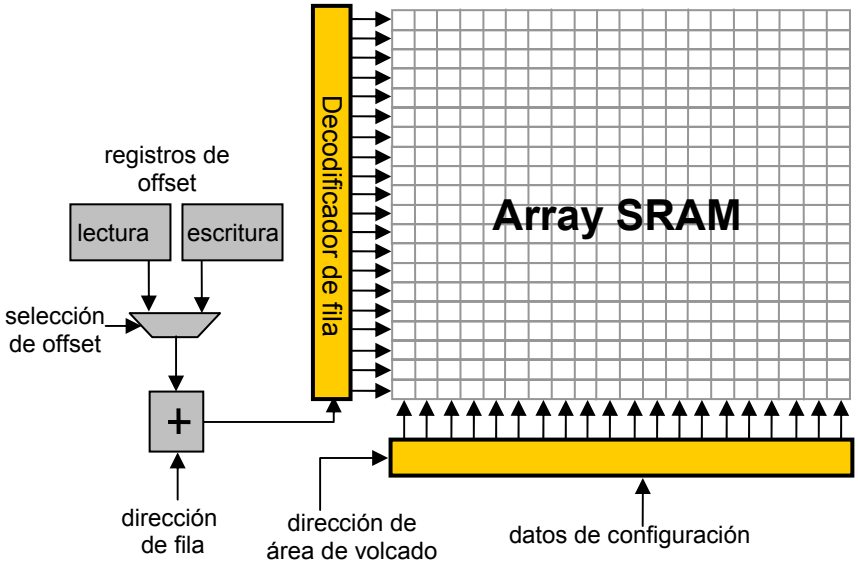


Figura 2.23. Arquitectura R/D FPGA.

Una configuración o tarea puede estar formada por una o varias filas. Para reubicar una tarea, cada fila se lee o escribe completa en una única operación a través del buffer especial de volcado. El direccionamiento se realiza proporcionando el número de fila dentro de una configuración, y un desplazamiento (*offset*) de fila (en lectura y escritura), que es el desplazamiento de la primera fila donde se va a leer o escribir la configuración respecto de la primera fila de la FPGA.

La figura 2.24, muestra un ejemplo de cómo se realizaría un proceso de defragmentación mediante reubicación fila a fila. Primero se copia cada fila de la configuración que se desea mover al buffer y después se copia de nuevo desde el buffer a la nueva ubicación de la fila. Las filas deben ser movidas desde sus posiciones actuales en la FPGA hasta las nuevas ubicaciones sin sobrescribir ningún dato necesario. Esto es evidente cuando la nueva ubicación de la configuración se solapa parcialmente con la posición actual, de modo que dependiendo del orden de movimiento de las columnas, se puede perder la información de una o más filas. Para mover una configuración hacia abajo se debe seguir un orden y empezar por la fila que esté más abajo (para el caso contrario empezar por la fila superior).

En el proceso de defragmentacion se usan los dos registros de *offset*: el registro de lectura se usa para almacenar el *offset* de la posición original de la configuración y el de escritura almacena la nueva posición para la configuración. La figura 2.24 muestra el funcionamiento del proceso de defragmentacion, donde en la fase (a) se usa un *offset* de lectura de 4, de modo que la información de la fila superior de la configuración que se desea mover hacia arriba se almacena en el buffer de volcado (b). En el paso (c) se usa un *offset* en el registro de escritura de 3. A continuación se incrementa la dirección del decodificador de fila, y el proceso continúa con la siguiente fila, como se muestra en (d) en la que se lee la segunda y última fila de la configuración y se almacena en el buffer de volcado en (e). Finalmente en (f) se escribe la última fila de la configuración que se esta reubicando y finaliza el proceso de defragmentación.

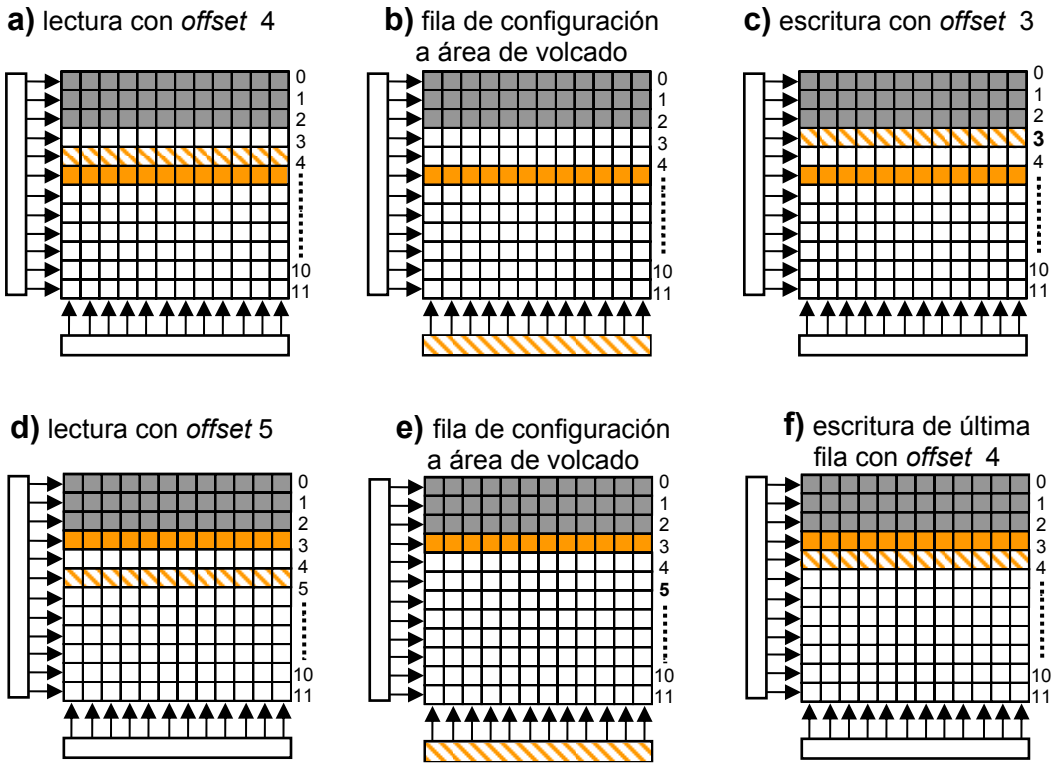


Figura 2.24. Proceso de defragmentacion en FPGA R/D.

Para realizar defragmentación se tiene que tener en cuenta el coste adicional de esta arquitectura, así como el elevado coste en tiempo para

reubicar tareas. La reubicación de tareas realmente está realizada sólo en una dimensión: una tarea dada se puede mover sólo a una fila diferente (arriba o abajo) pero no a una columna diferente (izquierda o derecha) por lo que la capacidad de reubicación está limitada.

2.2.2 Trabajo de O. Diessel

O. Diessel [DiEl01] trata la defragmentación a través de reubicación, y propone dos técnicas que abordan la defragmentación parcial del dispositivo, reubicando un subconjunto de las tareas que hay en ejecución.

El primer método denominado *Local Repacking* intenta minimizar el tamaño del área del dispositivo que necesita una reorganización de las tareas. Este método utiliza una búsqueda ordenada sobre la estructura en árbol que almacena la información del área libre de la FPGA, descrita en la sección 2.1.1. Con la búsqueda a través del árbol se identifican regiones que tienen área libre suficiente para ubicar la tarea en espera y se intenta reorganizar las tareas localmente en esa región. Para empaquetar las tareas dentro de la región se utilizan algoritmos de binpacking 2D como el algoritmo Sleator [Slea80].

El segundo método denominado *Ordered Compaction* restringe los movimientos de tarea y con la ayuda de un mecanismo de desplazamiento que, aplicado a las tareas residentes, posibilita juntar el espacio libre para futuras tareas que lleguen. Incluso propone algunas modificaciones HW para implementar estos mecanismos, pensados para arquitecturas de FPGA de una sola dimensión [BrDi01].

Esta técnica se realiza mediante el desplazamiento de las tareas en una dirección para hacer sitio, limitado a Horizontal (a la derecha) ó Vertical (hacia arriba). En la figura 2.25 se muestra un ejemplo en el que se intenta ubicar una nueva tarea de 6x5 BBRs y se realiza una compactación a la derecha.

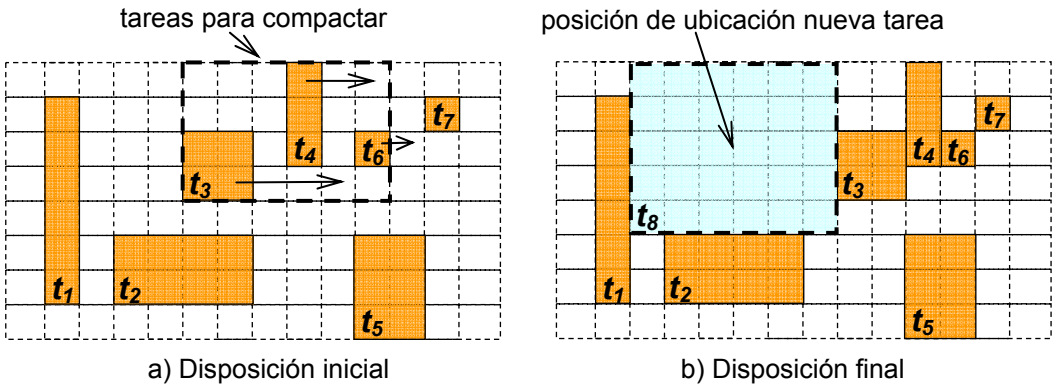


Figura 2.25. Ejemplo de la técnica *Ordered Compaction*.

Para la toma de decisiones se propone una estructura de datos donde se comprueba la posibilidad de efectuar una compactación. Para ello utiliza un grafo de visibilidad, donde cada tarea es un nodo, y hay enlace con otra tarea cuando hay visibilidad (solapamiento) en una dirección dada. La figura 2.26 muestra un ejemplo de visibilidad hacia la derecha.

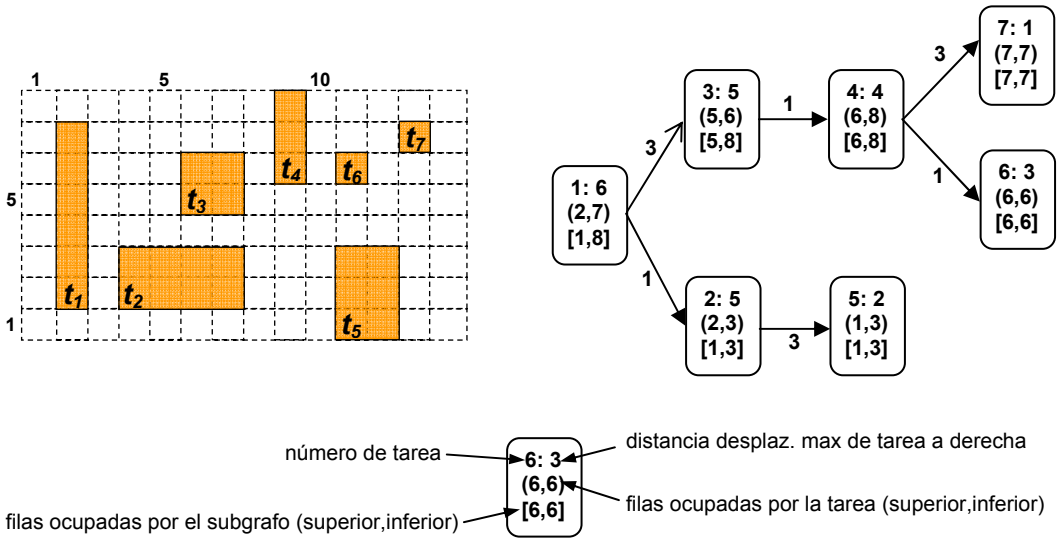


Figura 2.26. Grafo de visibilidad.

En el grafo de visibilidad cada tarea es un nodo del que salen enlaces a los nodos vecinos (tareas) más cercanos en la dirección de desplazamiento.

La figura 2.27 muestra los pasos planificados para compactar las tareas de una manera ordenada basada en la información proporcionada por el grafo de visibilidad correspondiente a la figura 2.26.

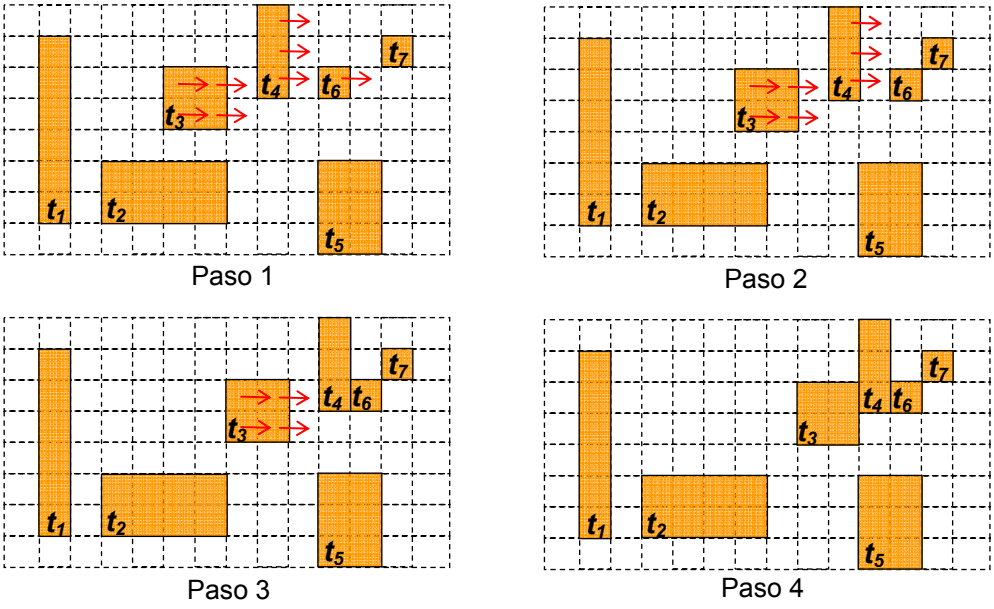


Figura 2.27. Fases de compactación de tareas hacia la derecha.

El trabajo de O. Diessel no aborda técnicas de defragmentación global ni preventiva, sólo de tipo urgente cuando una tarea entrante no encuentra sitio disponible. Tampoco utiliza ninguna métrica del estado de fragmentación resultante, limitándose a hacer sitio sólo cuando hace falta.

2.2.3 Trabajo de M. Handa

M. Handa presenta en [HaVe04b] una métrica de fragmentación y una estrategia de selección de MER basada en su métrica para obtener inserciones de tarea de calidad, con un entorno de gestión de tareas *online* en tiempo real. Los modelos de FPGA y tarea utilizados son los mismos ya vistos en el apartado 2.1.5, donde una celda puede representar un CLB o elemento lógico (LE).

La principal consideración es que las celdas vacías contribuyen de manera diferente a la fragmentación de acuerdo al número de rectángulos vacíos en su proximidad (una celda puede pertenecer a más de un rectángulo vacío) y que esta contribución es diferente en las direcciones vertical y horizontal. Dos celdas vacías se considera que están en la proximidad si se pueden conectar mediante una línea recta paralela a las caras horizontales o verticales de la FPGA y que además no esté cruzada por ninguna celda ocupada.

La figura 2.28 muestra una FPGA donde las celdas vacías A y B están en proximidad entre sí. Sin embargo, la celda A no está en proximidad ni con C ni con D y el número de celdas en la proximidad de A es 1 en dirección horizontal y 4 en la vertical. Para cada celda *c* se define *vx(vy)* como el numero de celdas vacías en la proximidad de *c* en la dirección horizontal (vertical).

El modelo de fragmentación propuesto cambia de valor de manera dinámica dependiendo del tamaño de la tarea que se pretende ubicar y la información de fragmentación se actualiza cada vez que se asigna o que finaliza la ejecución de una tarea.

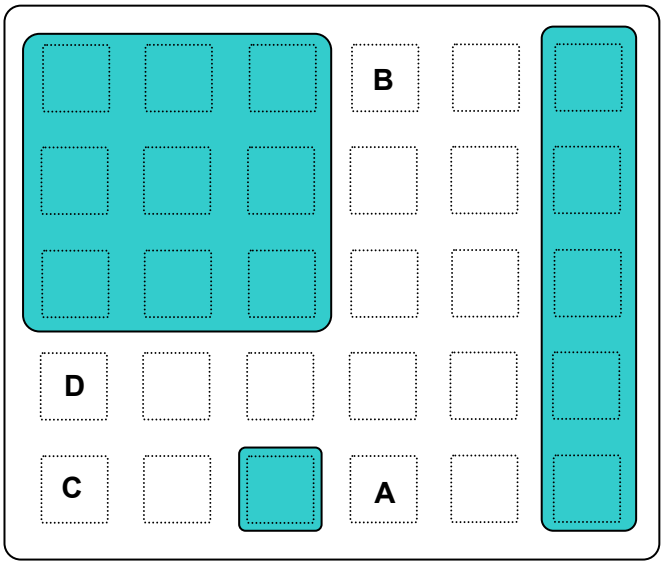


Figura 2.28. FPGA con celdas vacías en proximidad.

Se define la contribución de una celda vacía *c* a la fragmentación (CCF), como la cantidad de fragmentación con la que una celda contribuye a la fragmentación total de la FPGA, en un momento dado. Esta contribución a la fragmentación se calcula en las direcciones horizontal (2.1) y vertical (2.2). También se define la contribución total de una celda a la fragmentación (CCFT) como la suma en la dirección vertical y horizontal, y se calcula como en (2.3). Se define $\overline{L_x}$ como el doble de la dimensión horizontal media de las tareas que entran en la FPGA y $\overline{L_y}$ ídem para la dimensión vertical.

$$CCFc_x = \begin{cases} 1 - \dfrac{v_x}{\overline{L_x} - 1} & \text{si} \quad v_x \le \overline{L_x} \\ \\ 0 \ \text{en otro caso} \end{cases} \tag{2.1}$$

$$CCFc_y = \begin{cases} 1 - \dfrac{v_y}{\overline{L_y} - 1} & \text{si} \quad v_y \le \overline{L_y} \\ \\ 0 \ \text{en otro caso} \end{cases} \tag{2.2}$$

$$CCFTc = CCFc_x + CCFc_y \tag{2.3}$$

Se mantiene la información de la fragmentación actual del dispositivo con una matriz bidimensional de fragmentación (MF). La figura 2.29 muestra una matriz de fragmentación de una FPGA antes y después de asignar una nueva tarea. En el ejemplo la MF tiene 5 filas y 6 columnas de celdas *c*. La dimensión de la MF es la misma que la FPGA y cada celda vacía mantiene cuatro valores enteros *(vx, vy, lx, ly)*. El valor *vx* (*vy*) indica el número de celdas vacías en la proximidad de *c* en la dirección horizontal (vertical), y donde *lx (ly)* es la dirección de columna (fila) más a la izquierda (más abajo) de una celda vacía en la proximidad de *c*. Las entradas *vx* y *vy* ayudan a mantener la información de fragmentación de las celdas y *lx* y *ly* ayudan a la actualización de los valores de *vx* y *vy* después de que entre o salga una tarea en la FPGA.

Esta matriz de fragmentación se puede utilizar para calcular la contribución a la fragmentación de una única celda, o de un rectángulo vacío o incluso para el total de la FPGA. En la figura 2.29, se muestra una FPGA con 6*5 celdas y dos tareas en ejecución y el proceso de actualización de los valores *(vx, vy, Ix, Iy)* de las celdas afectadas después de insertar la tarea T_3. Después de insertar la tarea es necesario actualizar todos los valores de las celdas que están en la misma fila y columna: en la dirección horizontal, el valor *vx* de todas las celdas en la proximidad de la tarea, cambiarán a un nuevo valor *vx´* que refleje el nuevo número de celdas vacías en su proximidad. Lo mismo ocurre para *vy* en la dirección vertical.

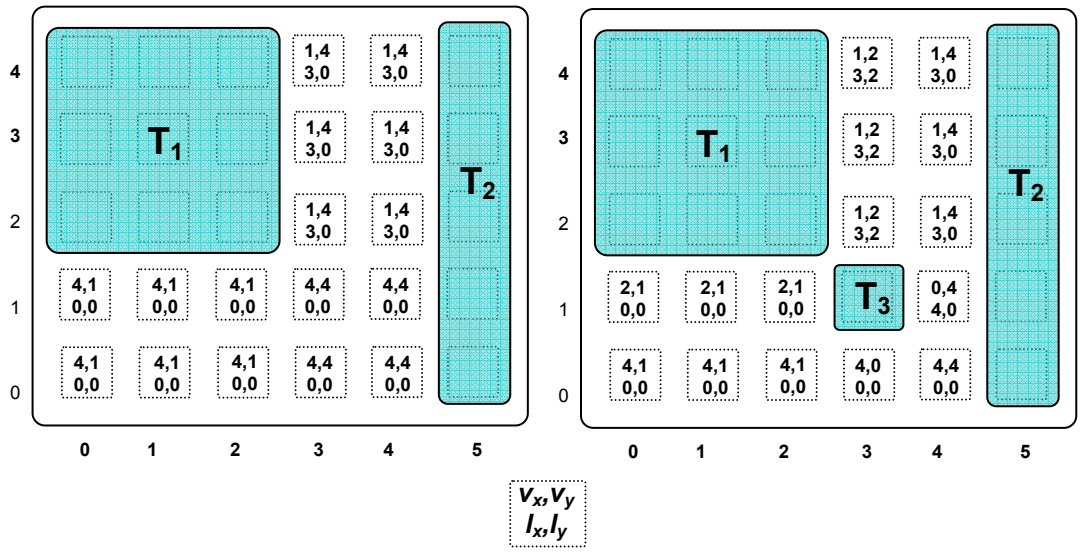


Figura 2.29. Ejemplo de matriz de fragmentación.

Este modelo de fragmentación se utiliza en la asignación *online* para obtener asignaciones de mejor calidad. El algoritmo de ubicación de tarea sigue utilizando el algoritmo de búsqueda de espacio libre comentado en el apartado 2.1.5, para encontrar la lista de rectángulos máximos vacíos en la FPGA, pero se utiliza el modelo de fragmentación para elegir el mejor candidato entre todos los MER que son suficientemente grandes para ubicar la tarea.

Este trabajo utiliza una matriz bidimensional para calcular el valor de fragmentación con una complejidad de gestión muy elevada.

2.2.4 Trabajo de H. Walder

H. Walder presenta en [WaPl02] un sistema reconfigurable en el que las tareas son independientes, no tienen restricciones temporales y no son reubicables: una vez que se han cargado en la FPGA, se ejecutan hasta su finalización. Se propone un algoritmo Best Fit que determina el grado de fragmentación de todas las posibles ubicaciones y selecciona aquella posición que minimiza el grado de fragmentación.

Para una FPGA con una distribución de tareas dada, se calcula el rectángulo más grande posible en el área libre. Este proceso se repite continuando con el próximo rectángulo más grande, hasta completar el total del área libre. El resultado es un histograma de áreas libres rectangulares. El histograma consiste en *i* clases, cada una con n_i rectángulos de tamaño a_i. El grado de fragmentación se calcula como:

$$F = 1 - \frac{\sqrt{\sum_i \left(n_i * a_i^2\right)}}{\sum_i \left(n_i * a_i\right)} \qquad (2.4)$$

El grado de fragmentación está acotado por 0 ≤ F< 1. Cuanto menor es F mayor es la posibilidad de que una futura tarea pueda ser ubicada en el dispositivo.

La figura 2.30 muestra una FPGA de 20*20 BBRs, con una tarea en ejecución. Se calcula el grado de fragmentación con la métrica propuesta en (2.4) cuando llega una nueva tarea de tamaño 5*5 BBRs, para dos posibles posiciones. En este ejemplo se seleccionaría la posición en (b) para ubicar T_2 ya que es la que proporciona menor grado de fragmentación cuando se calcula según (2.4).

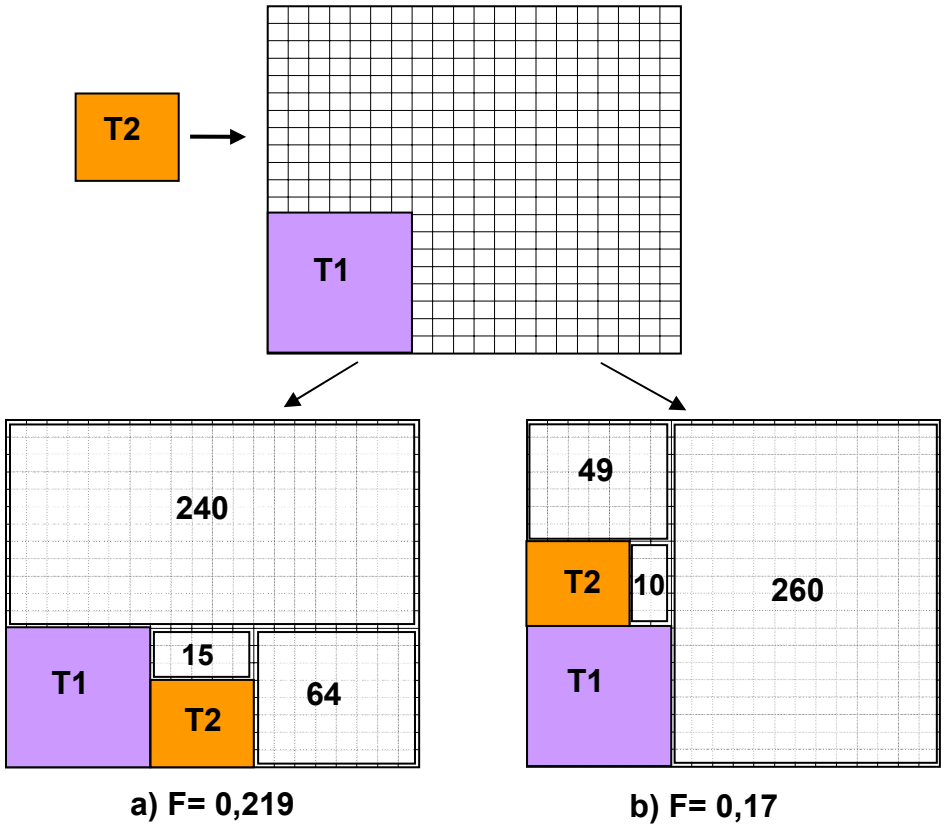


Figura 2.30. Ejemplo de cálculo de grado de fragmentación.

En la métrica propuesta por H. Walder, el grado de complejidad del cálculo es el mismo que el del algoritmo que calcula los rectángulos máximos O(n*logn), por lo que resulta demasiado alto. Además este método tiene que calcular la fragmentación para todas las posibles ubicaciones, con el coste computacional que ello supone.

2.2.5 Trabajo de A. Ejnioui

A. Ejnioui [EjDe05] presenta una versión simplificada de la métrica de fragmentación propuesta en [TSMM04], donde eliminan de dicha métrica el término que representa la adecuación de la forma de cada hueco. Define el factor de fragmentación f_i del hueco *i* que consiste en *k* celdas o bloques reconfigurables mediante ecuación (2.5):

$$f_i = \frac{1}{A}\sum_{j=1}^{k} a = \frac{ka}{N^2 a} = \frac{k}{N^2} \qquad (2.5)$$

donde *a* y *A* son el área de una celda vacía y del total del dispositivo, respectivamente, con un número total de celdas en la FPGA de N*N.

Y la métrica de fragmentación:

$$F = 1 - \left(\prod_i f_i \right) \qquad (2.6)$$

Algunos aspectos del problema de la fragmentación son considerados en este trabajo, pero los autores no muestran ninguna preferencia por alguna propuesta ni se presentan resultados experimentales. Además la métrica no tiene en cuenta la forma de los huecos y penaliza en exceso la formación de varios huecos.

2.2.6 Trabajo de J. van der Veen

J. van der Veen [VFMA05] utiliza una aproximación de tipo "*branch&bound*" (bifurcar y acotar) con restricciones, para realizar un proceso de defragmentación global que busca una disposición de módulos óptimo. Para simplificar la búsqueda se utilizan grafos de intervalos, como el mostrado en la figura 2.31. Esta clase de grafos han sido estudiados intensivamente en teoría de grafos [Golu80] y [Möhr85] y simplifican la búsqueda en lugar de utilizar complicadas descripciones geométricas. Para construirlos, se proyectan sobre cada eje de coordenadas, cada una de las cajas o tareas, de modo que una distribución en *d* dimensiones se convierte en *d* distribuciones unidimensionales. Las cajas están conectadas por una arista si sus intervalos proyectados en la dirección del eje de coordenadas tienen una intersección no vacía.

En la figura 2.31 se muestra un ejemplo del método de generación de los grafos de intervalos, y se puede observar en la zona superior izquierda una FPGA con cinco tareas en ejecución. A la derecha aparece la proyección sobre el eje Y, que sirve para generar el grafo G_2, mientras que debajo de la

FPGA aparece su proyección sobre el eje X, que sirve para generar el grafo G_1.

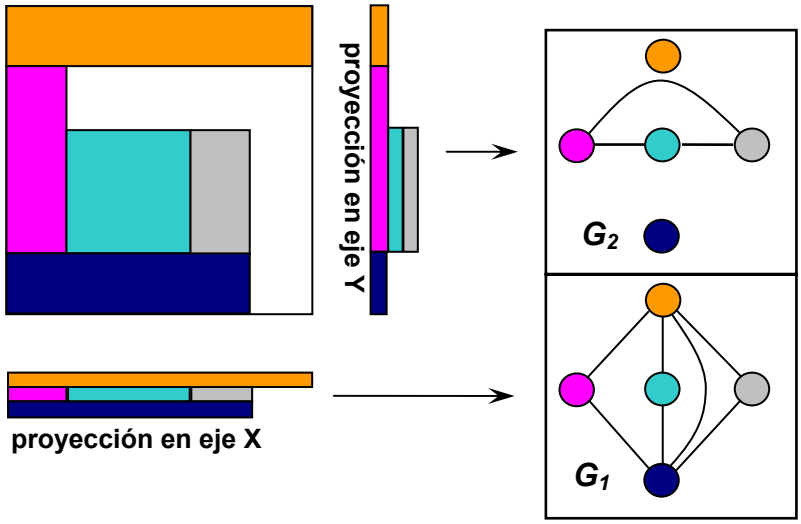


Figura 2.31. Proyección de tareas en los ejes de coordenadas para definir el grafo de intervalos en 2D con G1 y G2.

Este proceso requiere un alto coste temporal, aunque no se da ningún dato ni resultado en este aspecto. Además no se da ninguna información acerca de cuándo debe ser realizado el proceso de defragmentación.

2.2.7 Trabajo de H. Kalte

H. Kalte presenta en [KaPo05] un sistema para reubicar tareas, que reduce de manera significativa la cantidad de datos leídos desde la memoria de configuración de la FPGA, leyendo sólo las frames que contienen los datos de estado.

La arquitectura del sistema propuesto se muestra en la figura 2.32. En la parte de la derecha se muestra el sistema de asignación de ubicación a tarea con gestión 1D, que ya fue comentado en la sección 2.1.6 y presentado en [KKKP04]. En la parte central se muestran los cuatro bloques funcionales

principales: el Gestor de configuración, el Filtro de extracción del estado, el Filtro de inclusión del estado y el filtro REPLICA, junto con la base de datos, con los que se realiza el proceso de reubicación.

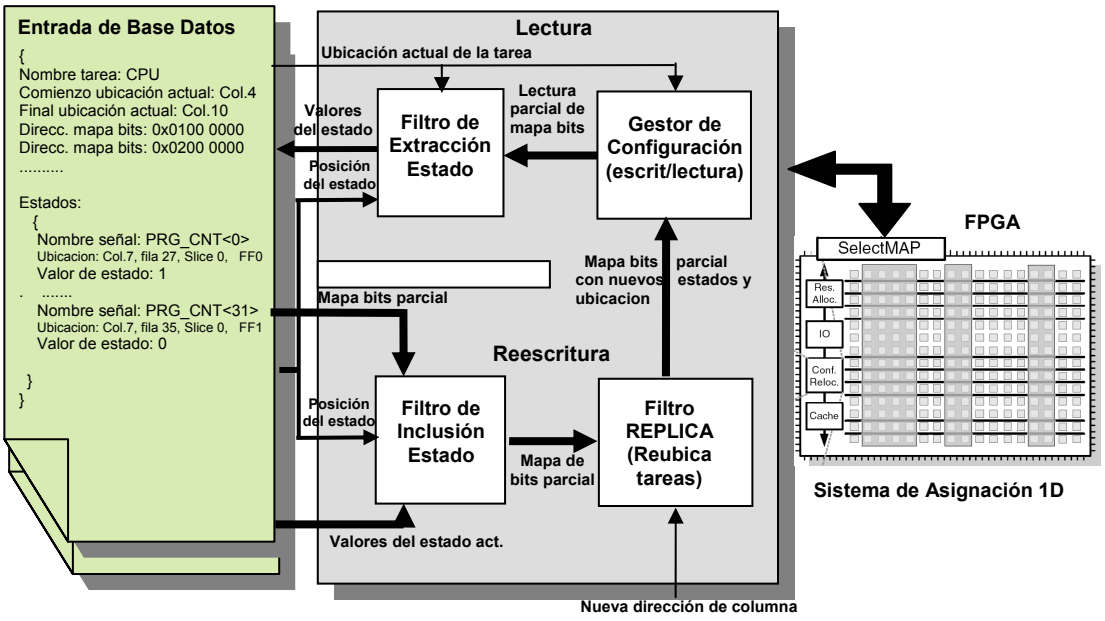


Figura 2.32. Arquitectura de sistema de reubicación de tareas sobre Virtex.

Este sistema utiliza el Puerto de acceso a la configuración SelectMap de la FPGA, a través del Gestor de configuración, para acceder a la información de determinadas columnas de la memoria de configuración, donde se hallan los contenidos actuales de RAM y registros del módulo que se desea reubicar. La dirección donde se encuentra cada tarea se extrae de la Base de datos, que debe estar actualizada en todo momento. Posteriormente el Filtro de extracción del estado filtra la información de este mapa de bits parcial, para extraer sólo la información del estado y posteriormente, manipular el contenido de registros y RAM del mapa de bits inicial del módulo que se desea reubicar. Después esta información del estado de la tarea a reubicar sirve para actualizar la base de datos.

A continuación se suspende la ejecución de la tarea, y se borra de la FPGA enviando un mapa de bits parcialmente vacío. Después se inicia el proceso de restauración de la tarea y se integra la información del estado actual de la

tarea (cuando se suspendió su ejecución) en el mapa de bits original de la tarea utilizando el módulo Filtro de inclusión del estado. Finalmente el filtro Replica reubica la tarea modificando la nueva dirección y el Gestor de configuración descarga la tarea a través del puerto de configuración de la FPGA para que se reanude su ejecución en el mismo momento que fue detenida.

Si comparamos esta solución con otras soluciones basadas en añadir un interfaz extra a todos los registros de estado de las tareas para su lectura/ escritura directa, como los presentados en [UHGB04] y [MNCV03], la principal ventaja es que no necesita ninguna estructura adicional que añadir a las tareas y como desventaja la baja eficiencia en la lectura de los datos del mapa de bits. Esta baja eficiencia es debida a que en los dispositivos Virtex la proporción de datos que contienen información del estado es de un 8% aproximadamente respecto del mapa de bits completo y que además esta información útil debe ser extraída posteriormente con el módulo Filtro de extracción del estado. Sin embargo, para acelerar el proceso de extracción de esa información no se leen todos los datos de configuración (sólo los que contienen la información del estado que pertenece a la tarea que se desea reubicar), y la extracción de dicha información no se realiza después, sino durante la lectura de los datos de configuración.

También presenta en [KoPK06] un algoritmo para realizar el proceso de defragmentación en tiempo de ejecución. Este algoritmo está orientado hacia arquitecturas heterogéneas 1D, como las Virtex-II ya que las tareas que usan memoria RAM no se pueden ubicar en cualquier posición de la FPGA. Utiliza el mismo entorno descrito anteriormente en [KaPo05] para reubicar tareas, utilizando por tanto la misma infraestructura de comunicaciones de bus horizontal combinado con una ubicación de tareas limitada a 1D.

Para cada posición viable de la nueva tarea, el algoritmo intenta reubicar todas las tareas que tienen intersección con el espacio que se necesita defragmentar y realiza una estimación de la sobrecarga de tiempo que se necesita para la reubicación de todas las tareas afectadas. Finalmente se elige la posición viable que genera la menor sobrecarga. Este tiempo dependerá del número de posiciones viables consideradas para la ubicación

de la nueva tarea. Por tanto el objetivo es la minimización del tiempo de defragmentación. Por ello el algoritmo se denomina *Partial displacement defragmentation* y se ha utilizado en combinación con el algoritmo de ubicación SUP Fit. Además, se muestran resultados experimentales donde se compara el algoritmo SUP Fit con defragmentación con otros dos que no la utilizan: SUP Fit normal y un Best Fit. Los resultados muestran que el tiempo necesario para procesar todas las tareas mejora cuando los tiempos de reconfiguración son pequeños comparados con los de ejecución.

Entre las limitaciones de este trabajo se encuentran que las tareas no tienen restricciones temporales para acabar su ejecución y que sólo se puede aplicar a arquitecturas 1D. Además, el algoritmo de defragmentación sólo se activa en caso de rechazo de tarea, cuando hay suficientes recursos, pero no reorganiza toda la FPGA, sino el mínimo de tareas suficientes para obtener espacio libre de forma contigua para alojar la nueva tarea.

2.2.8 Trabajo de G. Wigley

G. Wigley propone en [WiKe02b] una métrica que no necesita almacenar todos los MER disponibles. Para cuantificar la fragmentación del área sólo considera el MER de mayor área y define la *dimensión característica* del área como la longitud del lado menor del MER máximo. Sin embargo esta métrica sencilla no discrimina lo suficiente entre distintas situaciones, proporcionando los mismos valores para situaciones muy diferentes.

La figura 2.33 muestra dos ejemplos sencillos para el cálculo de la métrica. Las situaciones (a) y (b) muestran una FPGA de 20*20 BBRs con 5 tareas en ejecución dispuestas de diferente forma. En ambos casos la métrica proporciona el mismo valor aunque en (a) el MER máximo tiene mayor área que en (b).

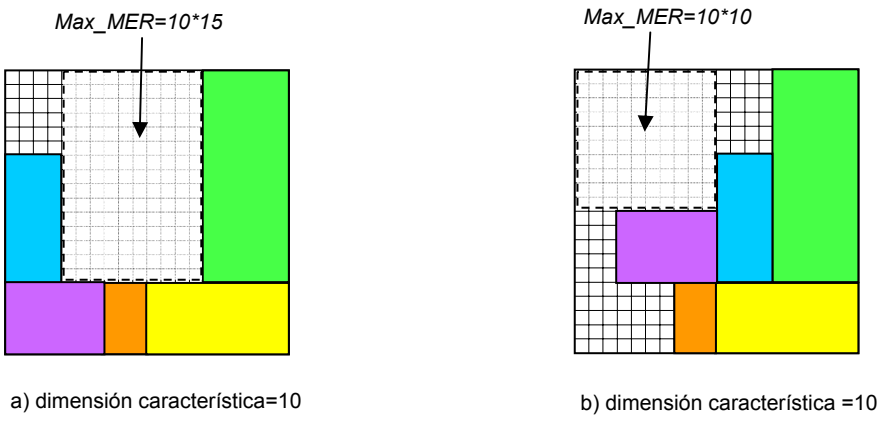


Figura 2.33. Estimación de la fragmentación con métrica de Wigley.

2.3 Valoración final y conclusiones

En este capítulo se ha revisado el trabajo de los grupos de investigación más relevantes en las áreas que serán tratadas en este trabajo de investigación, con el fin de poder realizar una mejor evaluación de las distintas soluciones aportadas.

En cuanto a la gestión del área libre, es reconocido que el enfoque de los MER de Bazargan en [BaKS00] pueden gestionar el área libre de una manera muy eficiente: si hay espacio libre para acomodar una nueva tarea, siempre puede ser localizado dentro de uno de los MER. Pero el número de los MERs en los que se debe realizar la búsqueda para encontrar el espacio libre es de $O(N^2)$, con N el número de tareas en ejecución. Como alternativa, otros enfoques basados en los NO-MER de Bazargan hacen una partición del área libre, dividiéndola en rectángulos no solapados (con objeto de reducir el número de rectángulos y por tanto la complejidad). Entonces pueden aparecer situaciones donde el espacio libre existente no se puede utilizar para ubicar una tarea, porque no está almacenado como parte de un único rectángulo, sino que se encuentra disperso entre dos o más rectángulos de espacio libre. Esta pérdida de eficiencia es el precio que hay que pagar por

una complejidad de búsqueda reducida de O(n*logn). En cualquier caso, ninguno de esos algoritmos intenta realizar una selección inteligente de la esquina exacta del rectángulo donde la tarea sería asignada, ya que la posición por defecto dentro del rectángulo siempre es la BL. Este hecho llevará probablemente a situaciones en la FPGA con peores valores de fragmentación, y con menos posibilidades de encontrar posiciones adecuadas para las tareas que demanden espacio libre en el futuro.

Walder muestra en [WaSP03] una versión mejorada del enfoque de los rectángulos NO-MER de Bazargan. Esto significa que, aunque la complejidad permanece baja, el área libre está todavía particionado y, aunque bajo ciertas condiciones se puede reparticionar, áreas libres contiguas que pertenezcan a rectángulos diferentes no se pueden considerar para ser asignados a una tarea.

Ahmadinia en [ABBT04] parte del trabajo de Bazargan pero gestiona el área ocupado en lugar del área libre y muestra que su enfoque para la selección de la ubicación tiene una complejidad de O(N), pero la heurística no intenta obtener una gestión eficiente del área libre. El único objetivo es minimizar los recursos utilizados en la interconexión que se debe realizar después de ubicar la tarea en el dispositivo. Este enfoque, donde solamente se consideran las comunicaciones punto a punto entre tareas, no está realmente enfocado a soportar multitarea HW real, y el coste temporal de la interconexión específica (*ad-hoc*) no parece adecuado para una asignación *online* o reubicación.

Finalmente Handa expone en [HaVe04] que su algoritmo basado en escaleras es equivalente al enfoque de los MER, con una complejidad de O(X*Y) siendo X e Y las dimensiones de la FPGA medida en BBRs, y que la heurística puede reducir hasta un 15% el número de localizaciones de la FPGA.

Dentro de la asignación de recursos y estructura del área libre, la mayoría de los trabajos que utilizan como referencia la estructura MER de Bazargan, heredan las principales desventajas de dicho planteamiento inicial: que la estructura de datos basada en MER tiene que ser actualizada cada vez que

se asignan o liberan recursos a una tarea, y dentro de los MER siempre se realizan la colocación de la tarea en posiciones BL, sin considerar cualquier otra posición que pudiese reducir la fragmentación en el futuro. Además, aunque cuando se realiza la búsqueda su orden de complejidad no es muy elevado, su actualización sí lo es.

En cuanto al problema de la fragmentación, la mayoría de los trabajos sólo tratan el problema a través de la reubicación de tareas. Para ello necesitan estructuras de datos complejas basadas en arrays bidimensionales con un alto coste de cálculo y actualización, más complejo de gestionar que las métricas de fragmentación presentadas en el capítulo 4. Concretamente, entre aquellos que proponen algún tipo de métrica para estimar la fragmentación, como el trabajo de M. Handa, se utiliza un array bidimensional para calcular el valor de fragmentación, que es más complejo de gestionar que las métricas de fragmentación presentadas en este trabajo de investigación. En la métrica propuesta por H. Walder el grado de complejidad del cálculo es el mismo que el del algoritmo que calcula los rectángulos máximos O(n*logn), por lo que resulta demasiado alto y A. Ejnioui presenta una métrica que deriva de una de las que se proponen en este trabajo de investigación, pero que tiene ciertas limitaciones, como se mostrará en el capítulo 4.

En este trabajo, se presenta una solución conjunta a todos los principales problemas presentados (Estructura del área libre, Asignación de recursos, y Gestión de la Fragmentación) basada en la utilización de una única estructura de datos, la Lista de Vértices. Además, a diferencia de la mayoría de trabajos presentados, para solucionar el problema de la fragmentación, se proponen no sólo medidas de urgencia, sino de carácter preventivo.

Capítulo 3:
Estructura del Gestor de HW básico

La propuesta presentada en este trabajo de investigación para la Gestión de HW Dinámicamente Reconfigurable (HWDR) mantiene la información del área libre disponible utilizando una estructura de Listas de Vértices, y decide dónde se ubican las tareas que van llegando seleccionando uno de los vértices de la lista. Para poder tomar esta decisión, se han definido unos modelos concretos de FPGA y tarea y se han desarrollado diversas técnicas de gestión y mantenimiento de la Lista de Vértices, a medida que las tareas van siendo procesadas.

3.1 Modelo de FPGA y de tarea

Las técnicas diseñadas para abordar la multitarea HW dependen en gran medida de la arquitectura interna y el sistema de configuración incorporado en la FPGA. Por tanto el punto de partida debe ser el modelo de FPGA y tarea considerados en el capítulo 1.

En nuestro modelo de FPGA se considera una arquitectura 2D, con un sistema de reconfiguración también en 2D y una organización homogénea, que se describe a continuación.

3.1.1 Modelo de FPGA

Como muestra la figura 3.1, se utiliza un modelo de FPGA en 2D: una malla homogénea de dos dimensiones que está formada por X*Y Bloques Básicos Reconfigurables ó BBRs, que serán usados en adelante como “unidades de área”.

Se supone que cada BBR está formado por un número de recursos suficiente para incluir elementos de procesamiento así como una interfaz estándar de E/S para la tarea, que permita la conexión a una malla de comunicación, que se usa solamente para comunicaciones entre tareas, nunca para comunicaciones locales intra-tarea. La infraestructura de comunicación genérica es necesaria para soportar una multitarea HW real, ya que permite que las tareas reubicables, situadas en cualquier posición, puedan realizar E/S sin realizar un rutado de nuevo, permitiendo incluso (con la ayuda de algunas técnicas) la reubicación de una tarea en mitad de la ejecución por motivos de defragmentación.

Este bloque básico BBR sería el tamaño mínimo de tarea permitido, aunque en general una tarea puede estar formada por un número arbitrario de dichos bloques básicos, pero siempre con forma rectangular, donde uno de los bloques básicos incorpora el interfaz estándar de E/S. Se ha optado por la

restricción en la forma rectangular (en vez de permitir formas arbitrarias) ya que es necesaria para permitir una gestión eficiente del área libre disponible.

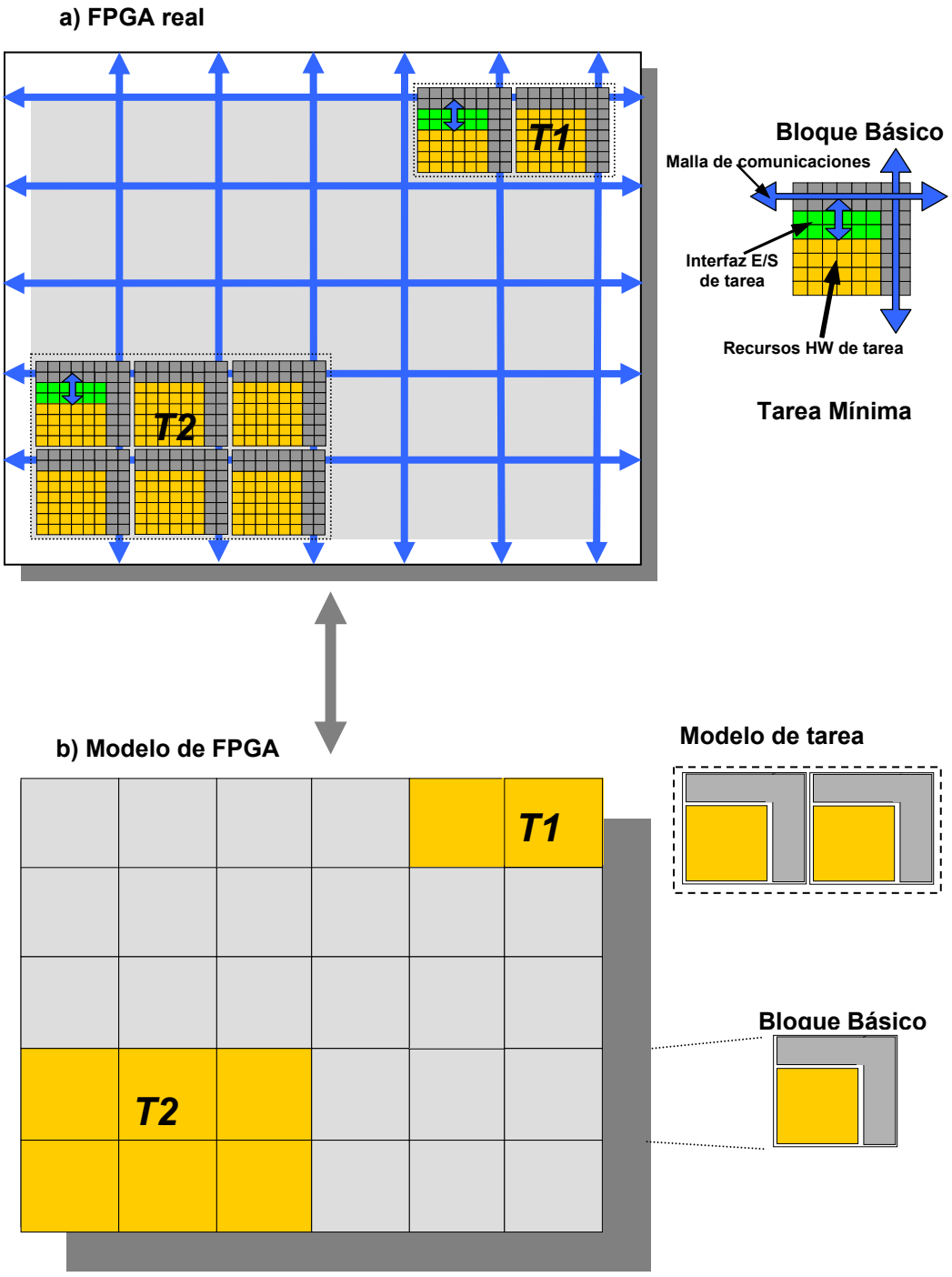


Figura 3.1. FPGA, modelos de tarea y E/S.

El mapa de bits de la tarea completa podría generarse con las herramientas de diseño disponibles en el mercado, y debe incluir además los recursos

necesarios para dar continuidad a la red de interconexión a través de la propia tarea (en el Anexo 1 se muestran algunas implementaciones prácticas que son compatibles con nuestro planteamiento) tal como muestra la figura 3.1.a.

Estas suposiciones permiten que las tareas sean completamente reubicables, es decir, que puedan ser ubicadas con desplazamientos (*offsets*) de columna y fila arbitrarios, en términos de unidades de BBR.

En adelante nuestra visión de la FPGA estará representada por el modelo de mayor abstracción representado en la figura 3.1.b, que obvia la presencia de la malla de comunicaciones.

3.1.2 Modelo de tarea

Como se describió en el punto 3.1.1, las tareas estarán definidas por el mínimo rectángulo que englobe todos los recursos HW que necesita la tarea. Por tanto no se considerarán formas arbitrarias, como las expuestas en la figura 1.9.

Cada tarea está definida por los siguientes parámetros:

$$T_i = \{ lx_i , ly_i , t\_ejec_i , \ t\_lleg_i , t\_max_i \} \quad (3.1)$$

Donde lx_i y ly_i indican el tamaño de la tarea (ancho y alto) medido en unidades de bloque básico BBR, t_ejec_i es el tiempo de ejecución, t_lleg_i es el tiempo de llegada y t_max_i es el máximo tiempo permitido para que la tarea finalice su ejecución. Todos los tiempos usados, excepto t_ejec_i, son absolutos (referidos al mismo origen de tiempo).

También se calcula, asociado a cada tarea, un tiempo de arranque límite como:

$$t\_lim_i = t\_max_i - (t\_conf_i + t\_ejec_i) \quad (3.2)$$

Donde t_conf_i, el tiempo necesario para cargar la configuración de la tarea, se estimaría de forma aproximada proporcionalmente a su tamaño:

$$t\_conf_i = k * lx_i * ly_i \quad (3.3)$$

El factor de proporcionalidad ***k*** dependería del tamaño del bloque básico elegido, de las características del interfaz de configuración (por ejemplo, el interfaz SelectMap de 8 o 32 bit para FPGAs Virtex descrito en [Xili03]), y de la tecnología de la FPGA. Para FPGAs 1D reconfigurables por columna ly_i es la misma para todas las tareas, y depende de la altura de la FPGA.

Finalmente, para cada tarea se define el **volumen de cálculo** de la tarea V_i como:

$$V_i = lx_i * ly_i * t\_ejec_i \qquad (3.4)$$

Este volumen representa la carga de trabajo que la tarea T_i demanda de la FPGA. En la figura 3.2 se muestra una FPGA (representada de acuerdo con nuestro modelo descrito en la sección 3.1.1 y dos tareas en ejecución. La tarea T_1 tiene una huella cuadrada con unas dimensiones de $\boldsymbol{lx_1}= \boldsymbol{ly_1}= 2$ y un tiempo de ejecución de 15 unidades. La tarea T_2 tiene una forma rectangular con una dimensiones de $\boldsymbol{lx_2}= 10$ y $\boldsymbol{ly_2}= 6$ y un tiempo de ejecución de 1. En la figura se muestra que aunque las dos tareas tienen un volumen de cálculo equivalente, demandan los recursos de la FPGA de modo diferente: T_1 sólo necesita 4 BBRs, y requiere de mayor tiempo para acabar su ejecución, mientras que T_2 sólo necesita un ciclo de ejecución pero demanda mayor cantidad de recursos de la FPGA (60 BBRs).

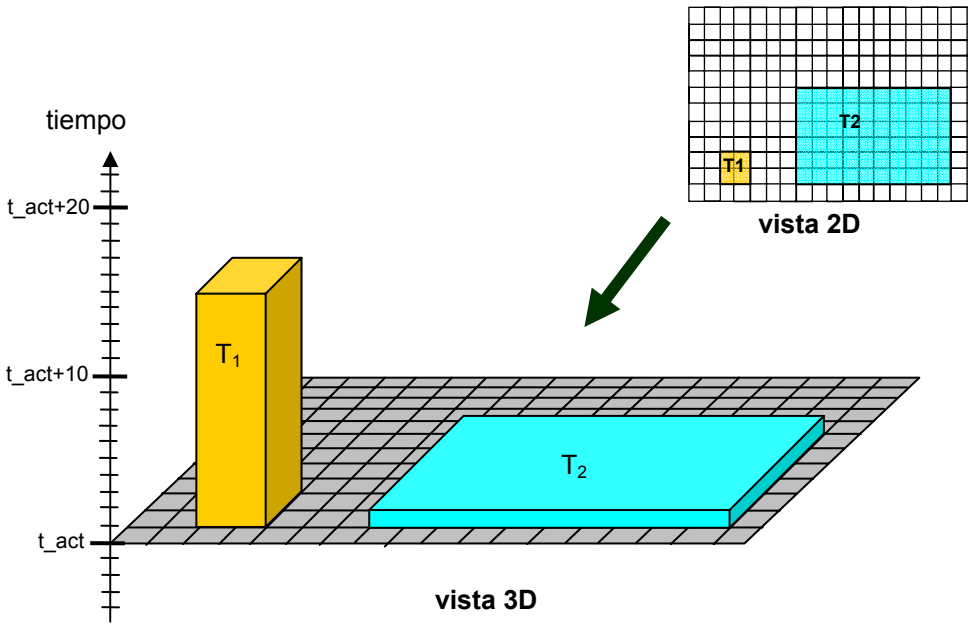


Figura 3.2. FPGA con dos tareas en ejecución.

3.1.3 Aspectos de planificación

Todos los parámetros temporales de cada tarea descritos en la sección anterior serán utilizados por el sistema de gestión de HW para su planificación, y en su caso para la asignación del inicio de ejecución de las tareas.

Cuando el sistema de gestión de HW considera en el momento actual *t_act* una tarea T_i, recién llegada o que estuviese en espera (por ejemplo por no haber hallado aún sitio para ubicarla), para asignarle un espacio en el dispositivo a fin de que pueda empezar su ejecución, se debe satisfacer la condición:

$$t\_act \leq t\_lim_i \tag{3.5}$$

Si no se puede satisfacer la condición (3.5), entonces la nueva tarea $\boldsymbol{T_i}$ será descartada para su ejecución en la FPGA.

Si aunque se cumpla la condición (3.5) la tarea $\boldsymbol{T_i}$ no puede empezar su ejecución por falta de disponibilidad de recursos HW en la FPGA, es posible demorar la planificación de la tarea mientras se siga cumpliendo todavía dicha condición. Otras posibles razones para la demora en la ejecución de la tarea son porque interese dejar paso a otra tarea más urgente, o porque en el futuro próximo finalice una tarea que deja libre una posición que es mejor según una heurística de asignación. Si no hay sitio para $\boldsymbol{T_i}$, el módulo encargado de la planificación, la almacena temporalmente en una cola de espera C_{esp}. Esta cola C_{esp} está ordenada de acuerdo al tiempo límite de arranque de cada tarea t_lim_i, para poder atender antes a las tareas que tienen un menor tiempo límite de arranque.

Cuando se dan todas las condiciones de espacio y de tiempo descritas anteriormente, se le asigna a la tarea $\boldsymbol{T_i}$ un tiempo de arranque t_inic_i que obviamente deberá cumplir:

$$t\_inic_i \leq t\_lim_i \tag{3.6}$$

Por otra parte, para un instante de tiempo dado *t_act*, cada tarea en ejecución T_j va a permanecer todavía en la FPGA por un tiempo:

$$t\_rest_j = t\_inic_j + t\_conf_j + t\_ejec_j - t\_act \quad (3.7)$$

Desde que llega una nueva tarea T_i en t_lleg_i hasta que se alcanza su tiempo de arranque límite t_lim_i, el sistema de gestión de HW dispone de un margen temporal para poder planificar la tarea t_marg_i, que se va reduciendo mientras no comienza su ejecución y en cada instante está definido por:

$$t\_marg_i = t\_max_i - (t\_conf_i + t\_ejec_i + t\_act) \quad (3.8)$$

Cuando empieza su ejecución en $t\_act = t\_inic_i$, el margen queda fijado a :

$$t\_marg_i = t\_max_i - (t\_conf_i + t\_ejec_i + t\_inic_i) \quad (3.9)$$

Las figuras 3.3, 3.4 y 3.5 muestran una FPGA con varias tareas en ejecución, y describen el proceso de planificación realizado por el módulo encargado de la planificación, por el cual decide dónde y cuándo se va a ubicar la tarea que llega. Para describir este proceso se ha representado el estado de la FPGA en tres instantes de tiempo diferentes. En cada una de las tres figuras se muestra el conjunto de tareas que se encuentra en ejecución caracterizadas por la tupla de parámetros definida en (3.1) junto a su tiempo estimado de configuración t_conf_i, y en el eje de tiempos los requisitos temporales de la tarea. Se supone que las tareas T_1 a T_6 han comenzado a ejecutarse en el momento de su llegada. En el interior de cada tarea aparece su t_rest_i correspondiente.

Cuando llega una nueva tarea T_N ={5,13,16,100,135} (figura 3.3), en el instante de tiempo actual *t_act=100*, no hay espacio libre suficiente, y el sistema de gestión de HW la envía a la cola de tareas en espera, porque se cumple la condición (3.5). Se calcula t_lim_N =135-21=114.

En la figura 3.4, en *t_act=109* la tarea T_5 finaliza su ejecución. El sistema de gestión de HW comprueba que se cumplen todas las condiciones de espacio y tiempo para que la tarea T_N pueda empezar su ejecución. Como estas condiciones se cumplen porque existen suficientes recursos HW disponibles y se cumple la condición (3.5) ya que 109<114, la tarea T_N empieza su

ejecución. En este momento el tiempo que T_N debe permanecer en ejecución es: $t\_rest_N = t\_conf_N + t\_ejec_N = 21$.

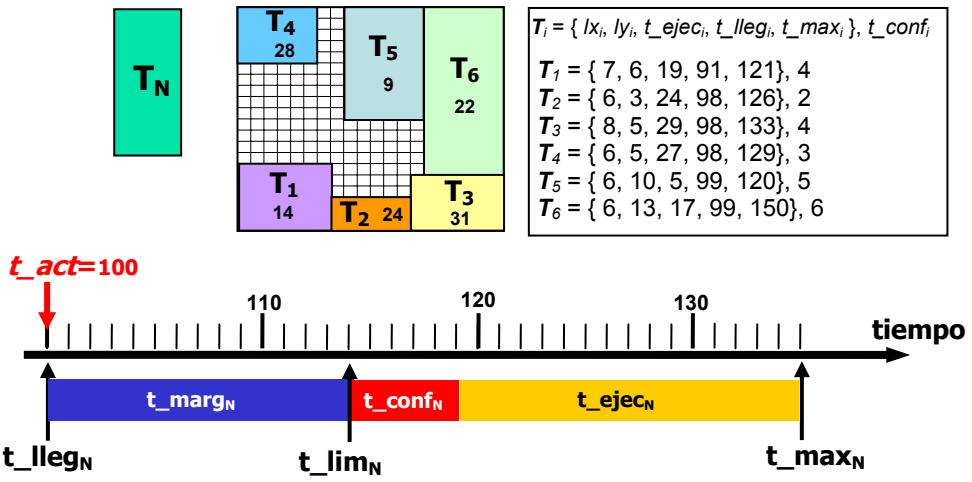


Figura 3.3. Planificación de tareas: fase (a) llegada de nueva tarea.

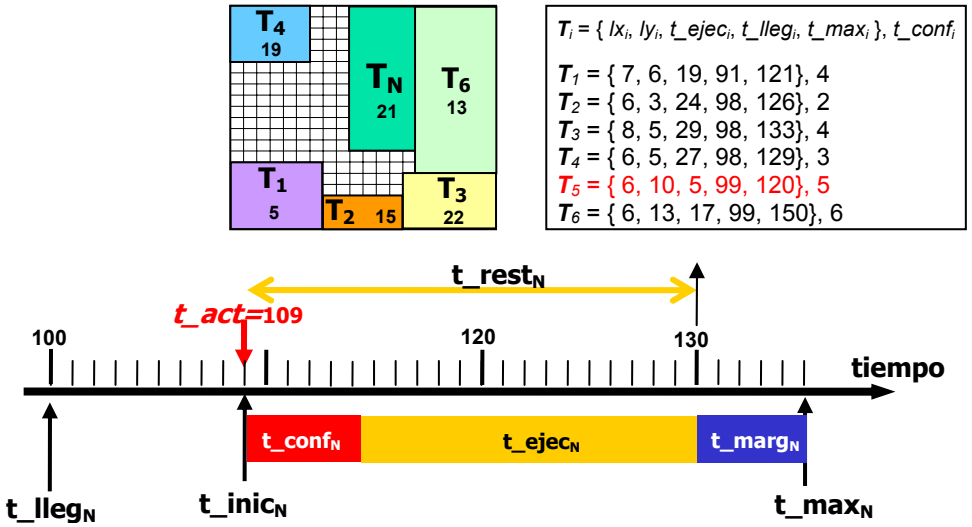


Figura 3.4. Planificación de tareas: fase (b) T_N empieza su ejecución.

En la figura 3.5 donde t_act=114, el t_rest_N correspondiente a T_N ha disminuido hasta 16, mientras que el módulo encargado de la planificación detecta que T_1 ha finalizado su ejecución cuando su t_rest_1 es cero. En ese nuevo estado, el sistema de gestión puede volver a utilizar el espacio que

ocupaba la tarea T_1 para la ejecución de otras tareas que lleguen en el futuro (o que estuviesen en espera por falta de recursos).

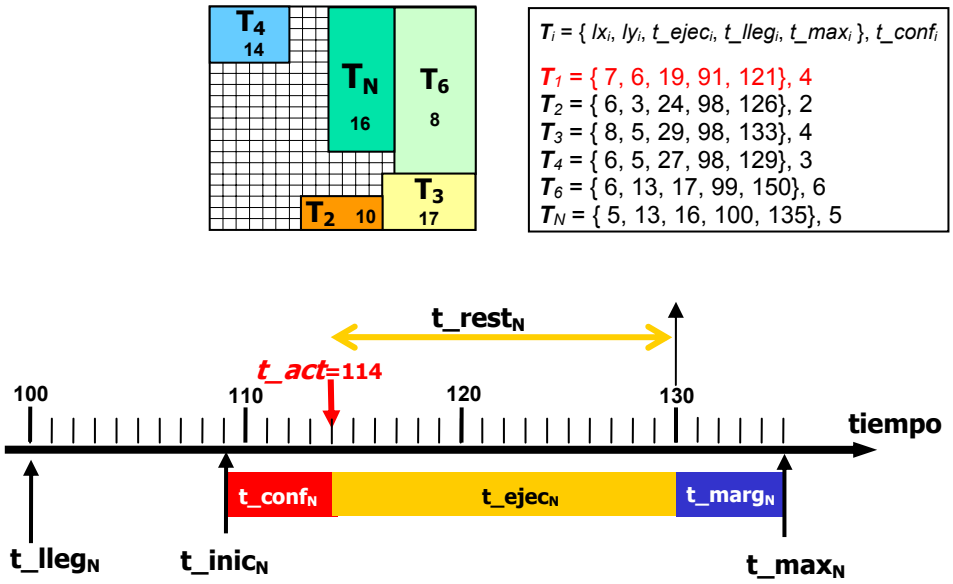


Figura 3.5. Fases de planificación: fase (c) T_N continúa su ejecución.

3.2 Modelado del área libre basado en Listas de Vértices

Para poder realizar una gestión efectiva del HWDR, el área libre del modelo de FPGA planteado en la sección 3.1.1, y mostrado en la figura 3.1, deberá ser modelado mediante una estructura de datos que facilite la búsqueda de posiciones libres y que pueda ser actualizada de una manera rápida. El modelo que se ha elegido es lo que se ha llamado Conjunto de Listas de Vértives.

El Conjunto de Listas de Vértices (CLV) consiste en una o varias Listas de Vértices LV_i, una por cada fragmento de área libre (o "hueco"). Esta estructura de datos es una descripción geométrica del perímetro del área libre disponible, y es similar a otras estructuras de datos usadas en Sistemas de Información Geográfica (GIS, de *Geographical System Information*) [BKOS97], y para problemas de *bin-packing*, como es la carga de un

contenedor con un conjunto de cajas. Los problemas de *bin-packing* tienen numerosas aplicaciones en la vida real pues permiten modelar, entre otras situaciones, la carga de camiones con limitaciones de peso, la distribución de los anuncios de televisión en las desconexiones o paradas publicitarias, la asignación de canales en redes de comunicación celulares, la asignación de trabajos de producción, o el diseño de circuitos VLSI entre otros.

La figura 3.6 muestra un ejemplo de un CLV con una única LV para describir un único hueco disponible.

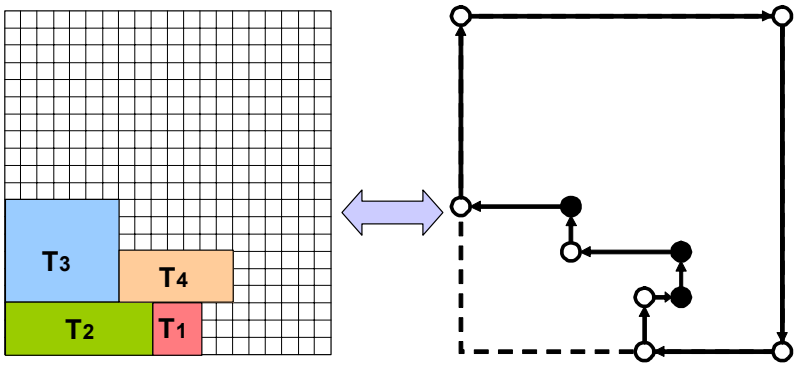


Figura 3.6. Estado de FPGA (a) y su Lista de Vértices asociada (b).

En la LV, algunos de los vértices, aquellos que son esquinas del perímetro, están marcados como **vértices candidatos,** ya que en ellos es posible ubicar una tarea (puntos blancos), mientras que otros no son considerados (puntos negros). Los candidatos pueden ser de diversos tipos, dependiendo del "tipo de esquina" que forma el candidato en la LV, y podrán ser del tipo **BL** (*bottom-left,* de abajo-izquierda), **BR** (*bottom-right,* de abajo-derecha), **TL** (*top-left,* de arriba-izquierda) y **TR** (*top-right,* de arriba-derecha).

Durante un proceso de ubicación, el algoritmo que busca un hueco para ejecutar la tarea recorre, en el sentido de las agujas del reloj, cada LV_i en el CLV y busca sólo en los vértices marcados como candidatos. Aquellos que permiten una ubicación de tarea viable, son los denominados **candidatos viables**.

Se considera una ubicación de tarea viable en una posición determinada, si los recursos necesarios para la ejecución de la tarea están dentro del espacio libre cuando se superpone la esquina BL de la tarea con el vértice candidato de la LV_i seleccionada, si el candidato es del tipo BL. Para otros tipos de vértice candidato la comprobación sería equivalente. Todos los detalles de la viabilidad en la ubicación de una tarea serán explicados en la sección 3.4.

3.3 Funcionamiento global del Gestor de Hardware

Un vez expuestos los modelos de FPGA, tarea y del área libre basado en Listas de Vértices, es necesario desarrollar un módulo Gestor que decida dónde y cuándo se debe ejecutar cada tarea.

Para ello, se establecieron dos objetivos para la realización del algoritmo de gestión de HW: que tenga un reducido tiempo de ejecución, es decir, que suponga una pequeña sobrecarga (*overhead*) en el tiempo de ejecución de la tarea, y que produzca una mínima fragmentación del espacio libre de la FPGA.

La figura 3.7 muestra y resume la operación básica de nuestro ***Gestor de HW***, que está formado por los siguientes módulos: el ***Planificador de tareas***, el ***Selector de vértice***, el ***Actualizador de Lista de Vértices,*** el ***Analizador de Lista de Vértices***, el ***Gestor de defragmentación,*** la ***Métrica de fragmentación*** y el ***Cargador/ Extractor de tarea***.

Estos módulos utilizan tres estructuras de datos importantes: la Lista de tareas en ejecución L_{ejec}, la Cola de tareas en espera C_{esp}, y el Conjunto de Listas de Vértices (CLV) que describe todo el espacio libre disponible.

Cuando se considera una tarea nueva $\boldsymbol{T_N}$ (en espera o que acaba de llegar), el *Planificador de tareas* realiza el proceso de planificación descrito en la sección 3.1.3. Para ello el *Planificador de tareas* llama al *Selector de vértices*, que comprueba si existe una posición viable donde la tarea se pueda colocar, consultando el CLV. Cuando es viable la ubicación, se elige un vértice entre

todos los candidatos viables de acuerdo a una heurística determinada, que serán expuestas en los capítulos 5 y 6. Después se ubica la tarea en el vértice elegido y el *Actualizador de Lista de Vértices* actualiza el CLV para reflejar la situación actual. En este caso se asigna a la tarea nueva T_N un tiempo de arranque t_inic_N, que debe satisfacer la condición (3.6).

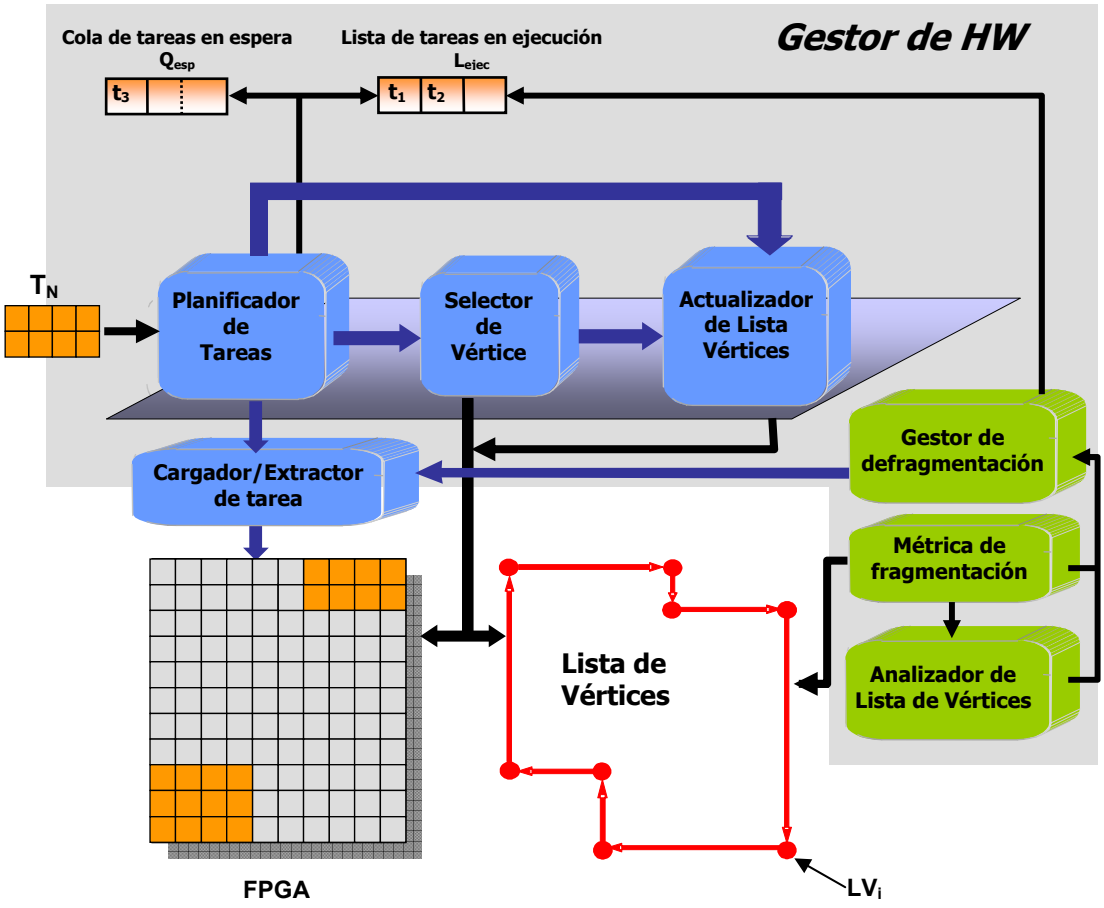


Figura 3.7. Estructura de Gestor de HW.

Si por contra, aunque se pueda cumplir la condición (3.5), no hay sitio para la tarea T_N, el *Planificador de tarea* la almacena temporalmente en la cola de espera C_{esp}. Esta cola C_{esp} está ordenada de acuerdo al tiempo límite de arranque de cada tarea t_lim_i, calculado como en (3.2). Si no se cumple (3.5) la tarea se rechaza.

Cuando el *Planificador de tarea* asigna una posición viable a la tarea T_N (reserva unos recursos HW concretos para su ejecución), entonces el

Cargador/Extractor de tarea se encarga de escribir el mapa de bits correspondiente en la memoria de configuración de la FPGA.

El módulo *Métrica de fragmentación* calcula el nivel de fragmentación del área libre de la FPGA para un estado dado, y será descrito en detalle en el capítulo 4. En todo momento, el estado de la fragmentación de la FPGA es monitorizado por el *Analizador de Lista de Vértices*. Cuando es necesario realizar un proceso de defragmentación, por alguna de las razones que serán expuestas en el capítulo 7, entonces se ejecuta el *Gestor de defragmentación*.

El *Cargador/Extractor de tarea*, cuando se realiza un proceso de defragmentación, realiza una lectura de la configuración y del estado actual (registros de estado) de la tarea, desde la ubicación original de la tarea en la FPGA. Estos datos de configuración y estado de la tarea son copiados posteriormente a la nueva posición asignada por el *Gestor de defragmentación*. Para confirmar que este proceso se puede realizar se tiene que comprobar que el coste del tiempo de reubicación es menor que el margen de tiempo disponible para la tarea en ejecución. El proceso de defragmentación será explicado en detalle en el capítulo 7.

Cuando una tarea finaliza y debe abandonar la FPGA, también se actualiza el CLV, y pueden aparecer diferentes situaciones especiales en el espacio libre resultante que deben ser tratadas de manera específica. Estas situaciones especiales serán descritas en detalle en la sección 3.4.

La figura 3.8 muestra el pseudocódigo del *Gestor de HW*. Antes de realizar la búsqueda de una posición viable para una nueva tarea, se intenta recuperar todo el espacio liberado por tareas que han finalizado su ejecución (bucle *Repeat*). Dentro del mismo bucle se intenta realizar la asignación de recursos a tareas que llegaron antes y que estaban esperando en la cola de espera C_{esp}, y se comprueba si hay alguna tarea en espera que ha dejado de cumplir la condición (3.5). Finalmente, en el último *If* se intenta asignar una posición de la FPGA a la nueva tarea T_N para que empiece su ejecución. En caso de no disponer de espacio libre suficiente, la nueva tarea se envía a la

cola de espera, hasta que se liberen recursos y se den todas las condiciones para que pueda empezar su ejecución.

Aunque no lleguen nuevas tareas, en cada paso de ejecución se comprueba si alguna tarea ha finalizado, por si puede empezar su ejecución, algunas de las tareas que estaban en espera en C_{esp}.

```
Repeat
{
      While TareaAcabaEjecucion( t_act)
      {
            ActualizarL_ejec(PlanificadordeTarea)
            ActualizarHuecos(ActualizadorDeListaDeVertice)
      }
      If HayTareasEnEspera(C_esp)
      For (pTarea=C_esp.begin();pTarea!= C_esp.end(); pTarea++)
      {
            If (TareaEspera_Timeout)
                  RechazarTareaParaEjecucionHW
            Else
            {
            If UbicaciónTareaEsperaViable(SelectorDeVertice)
                  {
                  InsertarTareaEspera(L_ejec)
                  EliminarTareaEspera(C_esp)
                  CargarTareaEsperaenFPGA(Cargador/ExtractorTarea)
                  ActualizarHuecos(ActualizadorDeListaDeVertice)
                  }
            }
      }
      t_act+1
}
Until (llega_nueva_tarea T_N)
If UbicaciónViableTareaNuevaT_N(SelectorDeVertice)
      {
      InsertarTareaNuevaT_N(L_ejec)
      CargarTareaNuevaenFPGA(Cargador/ExtractorTarea)
      ActualizarHuecos(ActualizadorDeListaDeVertice)
      }
Else
      InsertarTareaNuevaT_N(C_esp)
```

Figura 3.8. Pseudocódigo del Gestor de HW.

3.4 Actualización de Listas de Vértices

Para realizar la prueba de viabilidad en cada vértice candidato, el *Selector de vértice* consulta el conjunto de Listas de Vértices CLV. Para verificarlo, busca intersecciones entre aristas de cada LV_i del CLV (formadas por los

sucesivos pares de vértices de la LV) y las aristas de la tarea. Cuando el *Selector de Vértice* detecta una intersección entre aristas, rechaza el vértice candidato actual y continúa buscando el próximo candidato a través de la LV.

Los principales aspectos de la gestión del CLV son los procesos de ubicación y extracción de tareas que son detallados a continuación.

3.4.1 Ubicación de tarea

Una vez que se ha confirmado la ubicación de la tarea $\boldsymbol{T_N}$ en el hueco H_i como viable, entonces el *Selector de vértice* pasa el vértice candidato y los parámetros de la tarea al *Actualizador de Lista de Vértices* para que actualice la forma del hueco, modificando las LV_i afectadas. Para ello el *Actualizador de Lista de Vértices* desplaza algunos de los vértices existentes, y cuando sea necesario crea nuevos vértices y comprueba si estos nuevos vértices son candidatos.

La figura 3.9 muestra un ejemplo de cómo se realiza el proceso de comprobación de ubicación de tarea viable en los distintos vértices candidatos. En (a) aparece una FPGA de 20*20 BBRs con dos tareas en ejecución y en (b) el CLV correspondiente (formado por una única LV). En dicha figura aparecen en (c) numerados todos los posibles candidatos para realizar la ubicación de una tarea T_N que acaba de llegar a la FPGA. Para el caso de los vértices V_1, V_4, V_5 y V_6 la ubicación de la tarea sería **viable** porque el *Selector de vértice* no detectaría intersección entre las aristas de la LV y las de la tarea, cuando se superpone la tarea en cada uno de los candidatos. Esta ubicación viable indica que los recursos necesarios para la ejecución de la tarea están dentro del espacio libre definido por la LV. Para los otros vértices V_2 y V_3, no sería viable la ubicación porque hay intersección entre las aristas de la tarea y de la LV, indicando que una parte de los recursos necesarios para ejecutar la tarea en esa posición (aparecen con trama rayada) están siendo utilizados por otra tarea (si se ubica en V_2) o están fuera del límite de la FPGA (si se ubica en V_3).

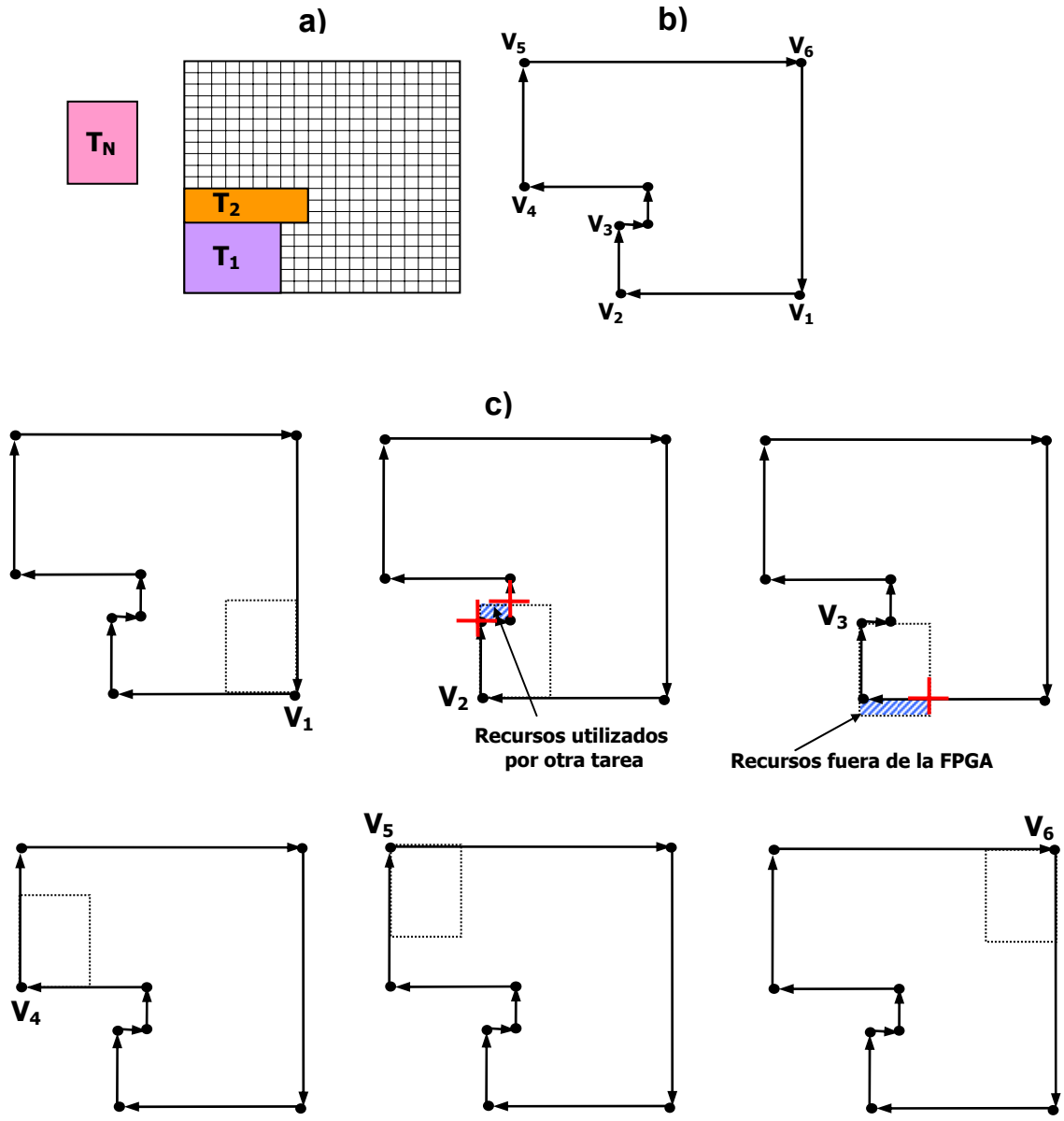


Figura 3.9. Comprobación de viabilidad en la ubicación de tarea. Estado de la FPGA (a), Lista de Vértices asociada (b) y comprobación de vértices (c).

Cuando es viable la ubicación de la tarea en el candidato seleccionado, se procede a actualizar el CLV, como se muestra en la figura 3.10.

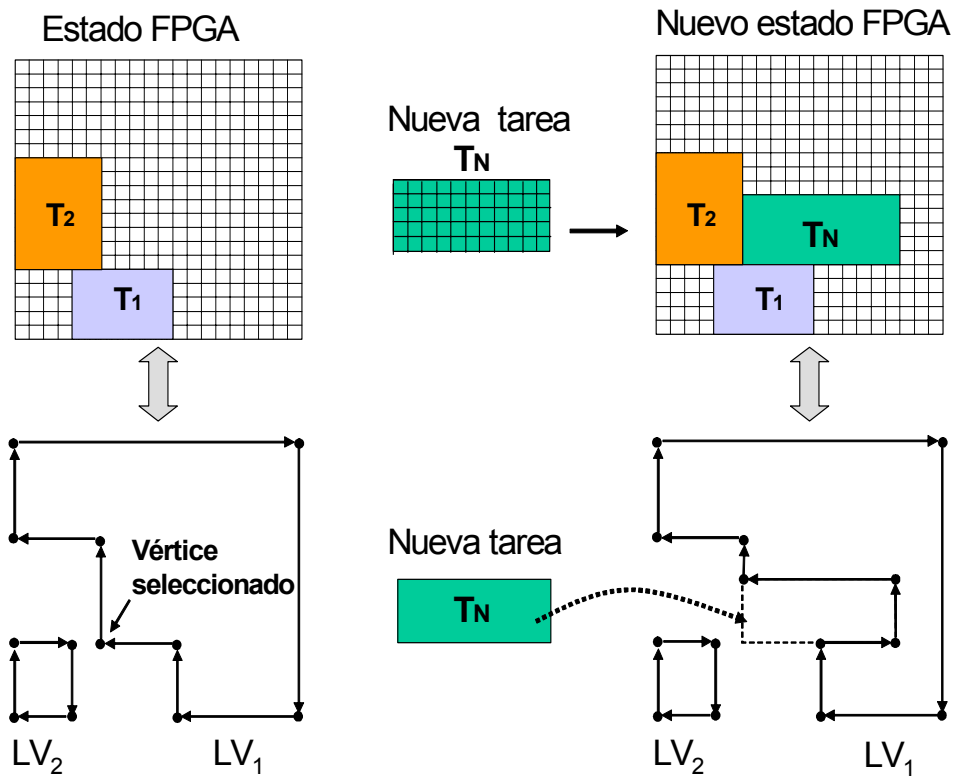


Figura 3.10. Ubicación de tarea y actualización del CLV.

El pseudocódigo del proceso de comprobación de viabilidad en la ubicación de una tarea, que se ejecuta en el módulo *Selector de vértice*, aparece en la figura 3.11. Su complejidad es $O(N^2)$ donde N es el número de tareas en ejecución en la FPGA en un momento dado. Como en el primer bucle *While* se tendría que recorrer la *Lista de Vértices* y comprobar en cada vértice candidato, si es un vértice viable para insertar la tarea T_N, la complejidad de esa búsqueda sería N. Como ya fue comentado en la sección 3.4, para la prueba de viabilidad en cada vértice candidato, el *Selector de Vértice* busca intersecciones entre aristas de cada LV del CLV y las aristas de la tarea y para ello es necesario recorrer la LV de nuevo en el segundo bucle *While*, con lo que la complejidad de la comprobación de viabilidad de ubicación es N.

```
UbicaciónViableTareaNuevaT_N(SelectorDeVertice)

For (pHueco(i)= begin();pHueco!= end(); pHueco++)
{
  If Hueco(i).Area>NuevaTareaT_N.Area;
  {
  CandidatoViable=false;
  While(NotfinVLi)And(NotCandidatoViable)
     {
      SuperponerTareaTN_enVerticeCandidato(i);
      RechazarVerticeCandidato=false;
      While(NotfinVLi)And(NotRechazarVerticeCandidato)
         {
         If IntersecciónAristaTareaAristaVL
               RechazarVerticeCandidato=true;
         Else
               CandidatoViable=true;
         AvanzarAristadeLV_i;
          }
       AvanzarVerticeCandidat;
      }
  }
}
```

Figura 3.11. Pseudocódigo del módulo de ubicación viable de tarea.

3.4.2 Extracción de tarea

El objetivo de este proceso es recuperar los recursos que utilizaba la tarea que finaliza su ejecución, para integrarlos dentro del hueco o huecos que definen el área libre disponible en la FPGA.

Cuando el *Planificador de tarea* detecta que una tarea ha finalizado su ejecución, extrae la tarea de $L_{ejec,}$ y llama al *Actualizador de Lista de Vértices* para actualizar el CLV. Entonces pueden aparecer diferentes situaciones dependiendo de si la tarea está en contacto o no con el perímetro del área libre de la FPGA definido por el CLV. Estas situaciones se deben tratar por separado y son descritas a continuación.

a. Tarea sin contacto con el perímetro. Cuando el *Actualizador de Lista de Vértices* detecta que la tarea saliente no es adyacente a ningún hueco, simplemente crea un nuevo hueco añadiendo una nueva LV al conjunto CLV. Esta situación se muestra en la figura 3.12.

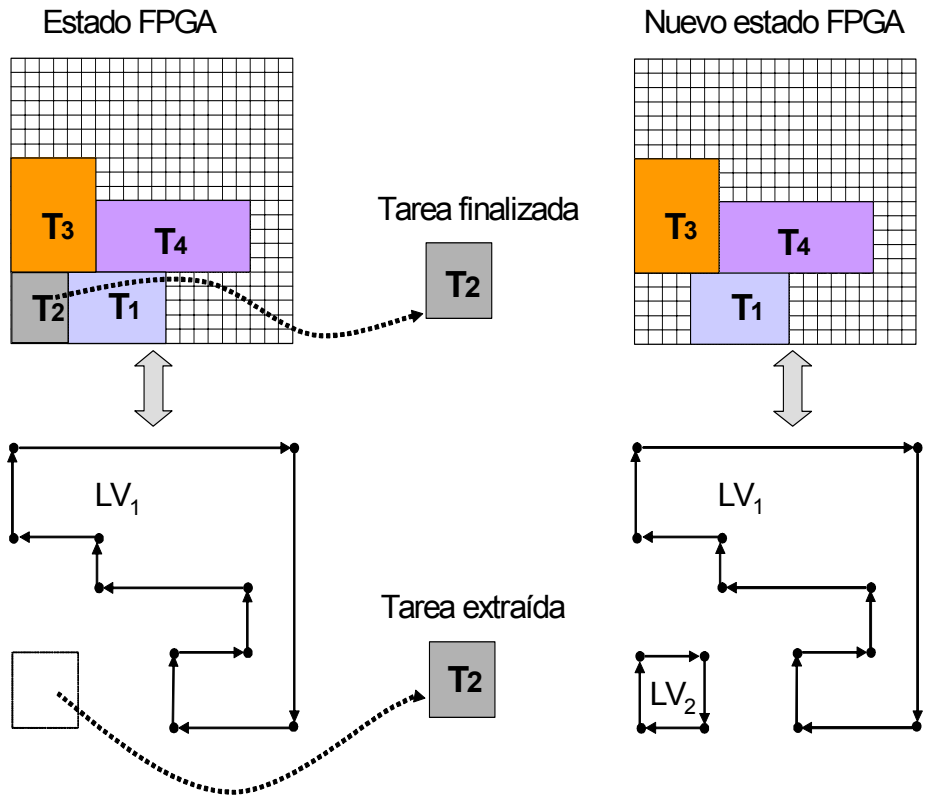


Figura 3.12. Extracción de tarea con un hueco nuevo separado.

b. Tarea en contacto con el perímetro. Si el *Actualizador de Lista de Vértices* detecta un punto de adyacencia entre cualquier arista del hueco y cualquier arista del perímetro del área originalmente ocupado por la tarea saliente, entonces añade el nuevo área libre al hueco, actualizando la correspondiente LV_i del CLV. Esta situación se puede ver en la figura 3.13.a y 3.13.b donde el *Actualizador de Lista de Vértices* detecta que la tarea saliente T_3 es adyacente al hueco definido por LV_1. Si la lista LV_2 no fuese también adyacente a T_3, LV_1 sería actualizado como muestra la figura 3.13.b y el proceso de actualización finalizaría.

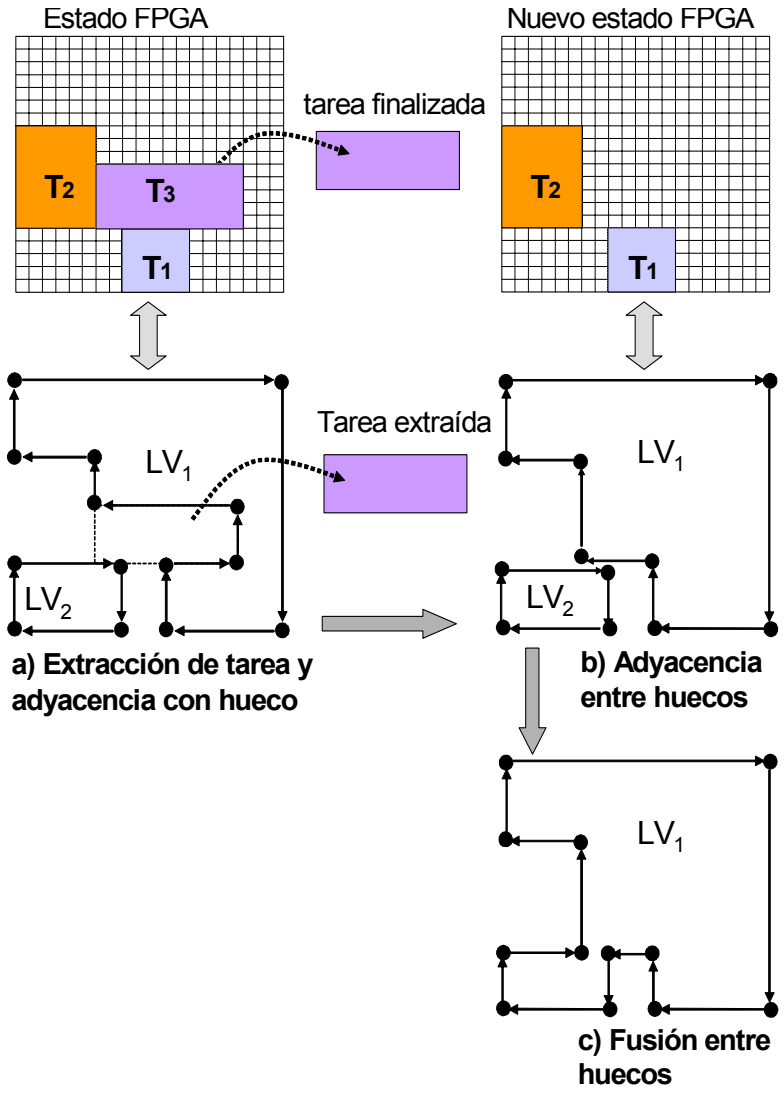


Figura 3.13. Extracción de tarea y fusión de huecos.

c. Fusión de huecos. Una vez que el *Actualizador de Lista de Vértices* ha fusionado el área donde la tarea saliente estaba mapeada, con cualquier hueco, entonces continúa comprobando a través del CLV para detectar otra posible adyacencia entre el hueco actualizado y otro hueco. Esta situación se muestra en la figura 3.13 en las fases (b) y (c), donde el *Actualizador de Lista de Vértices* detecta la adyacencia entre LV_1 y LV_2 y realiza el proceso de fusión dando como resultado un único hueco LV_1.

d. Islas. Algunas veces, después de una operación de extracción de una o varias tareas, otra tarea puede aparecer totalmente rodeada de área libre

como si fuera una isla. En estos casos el *Actualizador de Lista de Vértices* crea vértices virtuales que unen la envolvente del hueco con la envolvente de la tarea que forma la isla y viceversa, formando una única VL continua, como se muestra en la figura 3.14. Estas aristas virtuales (que unen vértices virtuales) no se consideran en la prueba de ubicación de tarea viable realizado por el *Selector de vértice*, aunque son necesarias para unir el perímetro de la isla al perímetro del área libre.

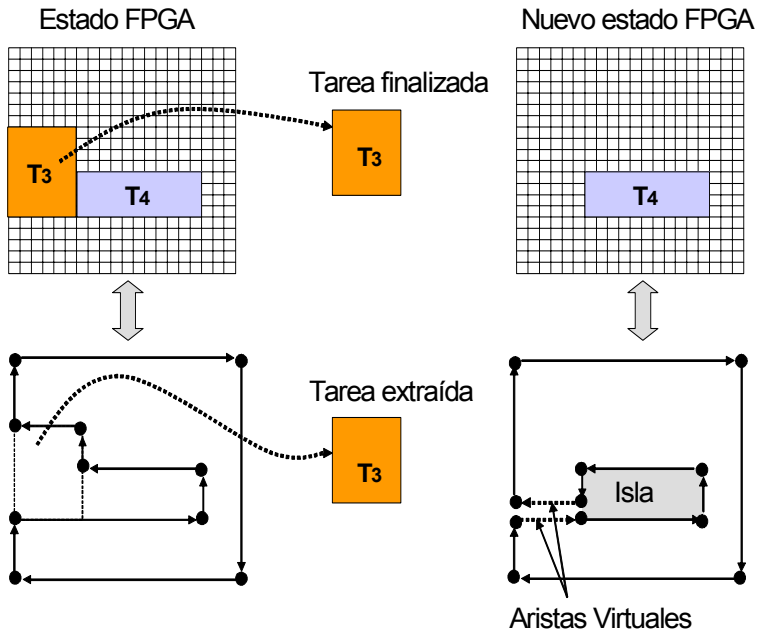


Figura 3.14. Gestión de islas.

La figura 3.15 muestra el pseudocódigo del proceso de *Actualización de Huecos,* que se invoca cuando el módulo de *Planificación de tareas* detecta que ha finalizado la ejecución de una tarea.

```
ActualizarHuecos(ActualizadorDeListaDeVertice)
For (pLV= CLV.begin();pLV!= CLV.end(); pLV++)
{
     pVertice=LVi.begin()
     While(NotfinLVi) and NotAdyacenciaEncontrada
     {
     If TareaSalienteEnPerimetro;
      {
            AreaHueco= AreaHueco+AreaTareaSaliente;
            ActualizaHuecoLVi;
            If DetectaIsla(pVertice)
                  MarcarAristasVirtuales;
            AdyacenciaEncontrada=true
     }
      If HuecoLViAdyacente_a_otroHueco
            FusionarHuecos;
       pVertice++;
     }
     If NotAdyacenciaEncontrada
     {
            Crear NuevoHuecoSeparado
            Añadir NuevoHuecoSeparado a CLV
     }
}
```

Figura 3.15. Pseudocódigo del módulo de extracción de tarea.

3.5 Valoración de la LV frente a otras estructuras

Los problemas de *bin-packing online* son muy difíciles de resolver mediante métodos de búsqueda exhaustiva debido a su complejidad NP (de Polinómico No determinista: conjunto de problemas que pueden ser resueltos en tiempo polinómico por una máquina no determinista). Algunos algoritmos tales como el de Diessel [DiWi99] o el de Bazargan [BaKS00] intentan reducir el tiempo de búsqueda con la ayuda de estructuras de datos complejas, como ya han sido mencionadas previamente. En ambos casos, para la ubicación de la tarea dentro del hueco, una vez que éste se ha seleccionado, se utiliza una versión simplificada eligiendo por defecto la posición BL.

3.5.1 Comparación con una solución basada en array

Cuando una tarea nueva T_N llega a la FPGA, un algoritmo FF clásico que utilizase un array bidimensional para realizar la gestión del área libre, realizaría una búsqueda exhaustiva, de derecha a izquierda y de abajo a arriba, para encontrar una ubicación viable para la tarea que llega. Esta heurística situaría la tarea entrante en la primera posición viable que encuentre.

Por otra parte, un algoritmo FF que utilice la Lista de Vértices (FF+LV), reduce la búsqueda y tratará de situar la tarea solamente en aquellas posiciones que son esquinas del perímetro del área libre de la FPGA.

En la figura 3.16 se muestra una comparación de funcionamiento entre estos dos métodos, mostrando en este caso que el área después de la ubicación de tarea está menos fragmentada con la solución que utiliza el CLV (a) que con un FF clásico en (b).

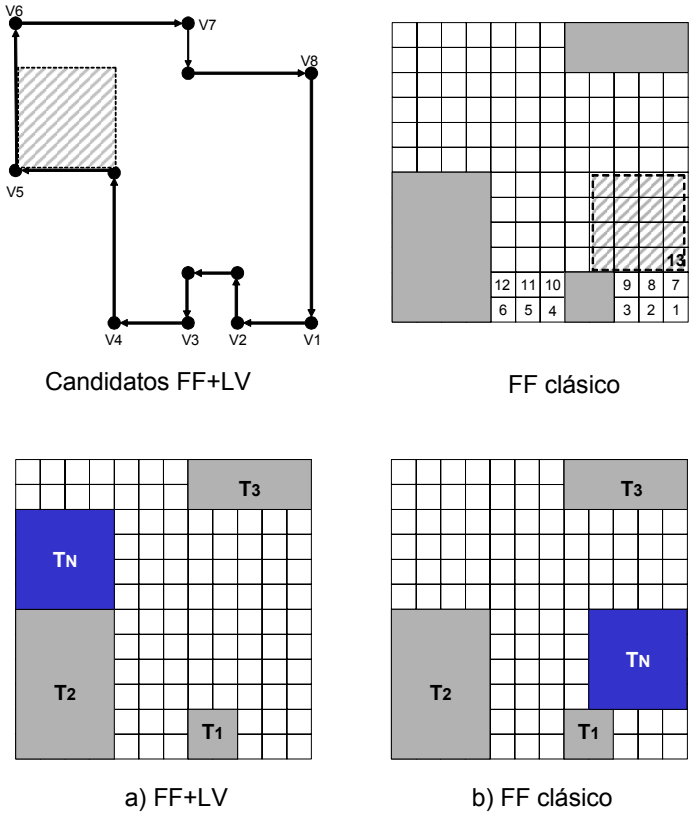


Figura 3.16. Comparación de heurísticas FF para ubicación de una tarea.

Cuando llega T_N, el algoritmo FF+LV sólo considera los candidatos viables que están en los vértices esquina y situará la tarea en V_5, mientras el FF clásico situará la tarea en el candidato 13º. El orden de complejidad del algoritmo FF clásico es mayor porque depende del tamaño de la FPGA en términos de BBRs. Para realizar la búsqueda de un candidato viable para la ubicación de la tarea, el algoritmo de búsqueda FF+LV tendrá que realizar el proceso de comprobación en cada uno de los vértices candidatos, que es proporcional a N (siendo N número de tareas en ejecución). Por lo tanto el uso de la Lista de Vértices acelera el proceso de ubicación de tarea, incluso aunque no se emplee otro tipo de heurística más sofisticada, y por lo cual se confirma la validez de este tipo de representación del espacio libre de la FPGA.

3.5.2 Comparación con soluciones basadas en rectángulos

A continuación comparamos la solución propuesta en este trabajo de investigación al problema de gestión del área libre, con otros enfoques revisados en el capítulo 2, y se demostrará que la Lista de Vértices resulta adecuada y favorable en términos de complejidad, actualización y calidad de las ubicaciones realizadas.

Como se ha comentado en la sección 2.3, la mayoría de los trabajos utiliza como referencia el trabajo de Bazargan, en el que se identifica el espacio libre de la FPGA a través de una lista de rectángulos MER. Por esta razón esta comparación se centrará directamente en el enfoque basado en MER como representativo de la mayoría de soluciones comentadas en la sección 2.1.

En el enfoque que se ha presentado, cada vértice de la LV puede ser considerado como una esquina de un MER (aunque es cierto que con ciertas formas extrañas pueden existir algunos MER no considerados en la LV). Por otra parte, ninguno de los enfoques previos basados en MER trata el problema de decidir cuál de las esquinas del MER es la más adecuada para

ubicar la tarea. De este modo, la esquina BL es considerada en general como candidato en todos los trabajos.

Por el contrario, la LV almacena como vértices sólo los más interesantes de los cuatro posibles candidatos de cada MER (abajo-izquierda, abajo-derecha, arriba-izquierda y arriba-derecha), aquellos que de forma natural llevan a situaciones de menor nivel de fragmentación. Entre estos, son elegidos sobre el resto aquellos que tiene una cualidad concreta de acuerdo a la heurística de selección elegida, tal como se explicará en los capítulos 5 y 6.

La figura 3.17 muestra un ejemplo de una FPGA con dos tareas, en la que se muestran dos de las alternativas para la gestión del área: la basada en MER y la basada en la LV.

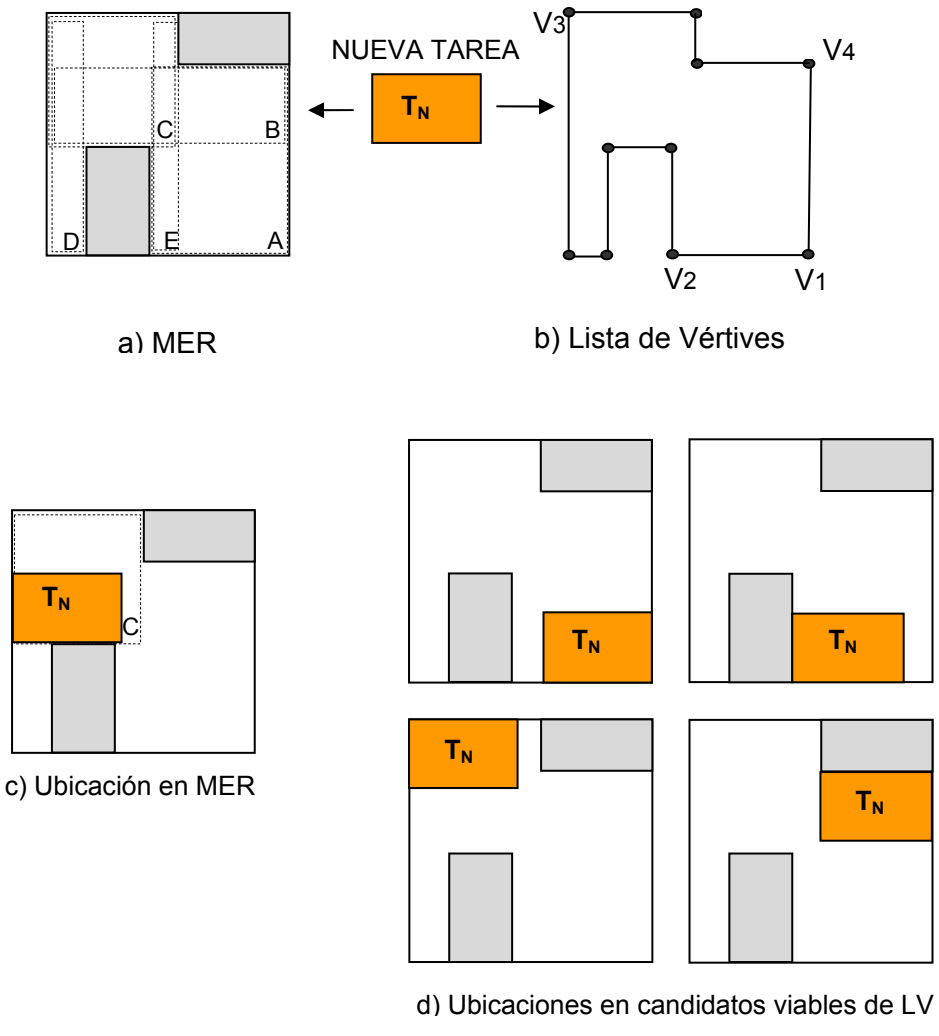


Fig. 3.17. Ubicación de tarea con MER (a) y Listas de Vértices (b).

En la figura 3.17.a aparece la FPGA con las tareas en ejecución, y la descripción del área libre realizada con 5 MER, etiquetados cada uno en la esquina BR de la A a la E. En (b) aparece la descripción realizada con la Lista de Vértices, donde aparecen numerados solamente los vértices candidatos viables para realizar la ubicación de la nueva tarea. En (c) aparece la tarea ubicada en el rectángulo C, que sería seleccionado si se eligiese una heurística Best Fit que busca en el espacio definido por los MER, pero en caso de utilizar otro tipo de heurística también podría ser insertada en el rectángulo A o B. Las posiciones consideradas utilizando la Lista de Vértices, dependiendo del tipo de heurística seleccionada, llevarían en todos los casos a situaciones donde el espacio libre resulta menos fragmentado como puede comprobarse en (d).

La complejidad de nuestro enfoque está determinada por el número de vértices de la LV que necesitan ser comprobados. El número total de vértices en la LV es de orden (N), siendo N el número de tareas en ejecución en la FPGA. El máximo número posible de vértices podría ser de 4*N+4, pero para acelerar la búsqueda se guarda una lista separada con los vértices candidatos solamente, cuyo número máximo (considerando todos los tipos de vértices que pueden ser candidato: BL, TR, TL o TR) es de 2*N+4. El máximo número posible de N sería para tareas de tamaño mínimo, igual a un bloque básico BBR. Pero la diversidad en el tamaño de las tareas y las características de la LV, mantienen el número real de vértices de la lista por debajo de ese máximo.

Como muestra la figura 3.18, se han realizado experimentos con una FPGA de 100*100 BBRs (con 10.000 posibles ubicaciones diferentes) y gran variedad de CLVs tomadas de los lotes de tareas usados en los distintos experimentos mostrados en los capítulos 5 y 6 (con el número de tareas en ejecución desde 1 a 14) que han mostrado que el número de vértices candidato corresponde aproximadamente a N+4. Esto significa que en la práctica cuando se utiliza la Lista de Vértices junto con una heurística simple de tipo First Fit se reduce el número de vértices candidato real en la LV, y por tanto la complejidad de la búsqueda, en aproximadamente un 50% respecto al máximo número posible. Estos experimentos muestran, por ejemplo, que

para un número medio de tareas en ejecución de N=7,5 y una ocupación media en la FPGA del 51%, el número de localizaciones consideradas es solamente de un 2,8% del total del tamaño de la FPGA, y un 5,5% del total de las posiciones libres disponibles.

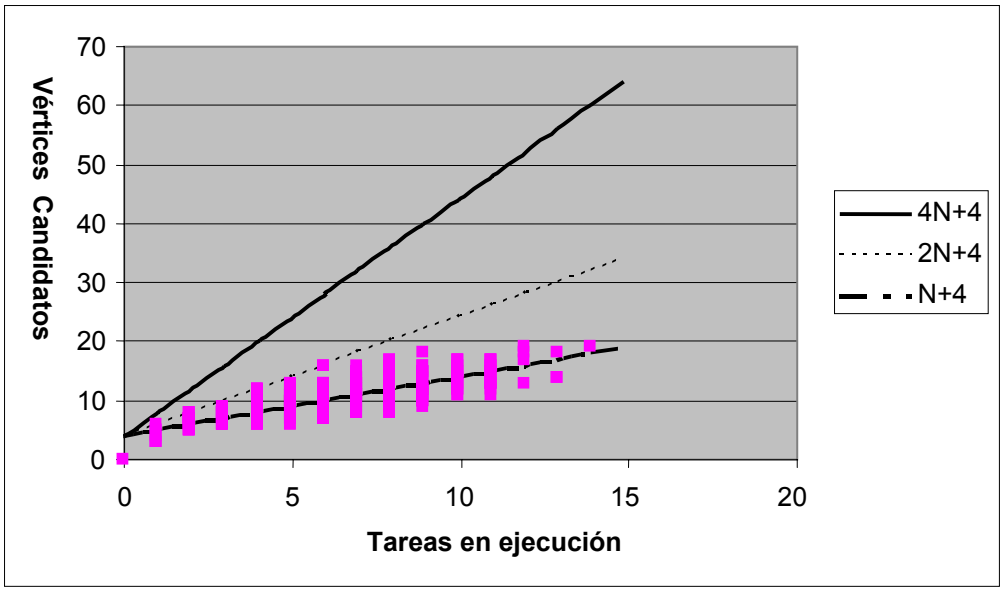


Fig. 3.18. Vértices candidato en el CLV con el número de tareas en ejecución.

Este tipo de experimentos se han repetido con el mismo tipo de FPGA y el CLV, pero empleando heurísticas de asignación de espacio más elaboradas (como las que se mostrarán en los capítulos 5 y 6), y la proporción entre tareas y número de candidatos se mantiene, aunque obviamente, se obtiene un mejor rendimiento en términos de ocupación y volumen de trabajo ejecutado.

Se puede concluir que el enfoque de Lista de Vértices tiene una calidad en la asignación de espacio libre a una nueva tarea, semejante al del enfoque con MER solapados, pero manteniendo una complejidad similar a la de otros enfoques más simples y menos eficientes, basados en rectángulos no solapados.

Capítulo 4:

Estimación del estado de fragmentación del área libre

De los dos tipos de fragmentación comentados en la sección 1.2, interna y externa, en nuestro modelo de gestión podemos considerar que, dado que usamos elementos básicos de tamaño muy pequeño (los BBRs), la fragmentación interna será despreciable. Por tanto, será sólo la externa la que estimaremos y gestionaremos para obtener un aprovechamiento óptimo de los recursos de la FPGA.

Aunque uno de los planteamientos iniciales para el desarrollo del *Gestor de HW* era la utilización de técnicas de elección de la ubicación que generasen de forma natural bajos niveles de fragmentación, el problema de la fragmentación aparece de manera ineludible por la propia naturaleza del entorno dinámico de entrada y salida de tareas.

Para técnicas de gestión simple basadas por ejemplo, en varias particiones de tamaño fijo de igual tamaño como en [MeLJ98], diferente tamaño como en [RMMS08], o incluso en particiones arbitrarias, aplicadas a arquitecturas comerciales con un sistema de reconfiguración 1D, tales como las Virtex de Xilinx, la fragmentación externa se puede detectar e incluso gestionar fácilmente. Este problema es similar al que aparece con la fragmentación de memoria en entornos multitarea software.

Para modelos de arquitecturas más complejos, tales como los reconfigurables en 2D, se deben usar técnicas más sofisticadas, como las revisadas en la sección 2.2, realizando un seguimiento del espacio libre disponible, para poder gestionar la FPGA de una manera eficaz. Para este tipo de arquitecturas la estimación del estado de fragmentación a través de una métrica precisa es una cuestión de gran importancia.

Aunque para 1D la estimación de la fragmentación es un problema lineal, para 2D se convierte en uno geométrico. La principal característica de una buena métrica de fragmentación debería ser su habilidad para detectar cuándo el área libre de la FPGA es más o menos apto (en función principalmente de su geometría) para acomodar las futuras tareas que lleguen, es decir, tiene que detectar si está organizado eficiente o ineficientemente, y proporcionar un valor cuantitativo para esa organización. La métrica debe separar la estimación de la fragmentación de la del grado de ocupación o de la cantidad de área libre disponible. Por ejemplo, un estado de una FPGA con un alto grado de ocupación pero con todo el área libre concentrado en un rectángulo único y casi cuadrado, no se puede considerar como fragmentado como lo hacen muchas de las métricas descritas en el capítulo 2. Además debe ser simple de calcular, por lo que resultan desaconsejables y menos prácticas las soluciones basadas en MER, que utilizan algunas métricas mostradas en dicho capítulo.

Esta métrica se puede utilizar con propósitos diferentes: primero, como función de coste cuando se tienen que tomar decisiones relativas a la búsqueda de ubicación de tareas. El uso de la métrica de fragmentación como función de coste garantizaría, en principio, futuros estados de la FPGA con menores niveles de fragmentación (para el mismo nivel de ocupación),

que llevarían a mayores probabilidades de encontrar espacio contiguo para acomodar una nueva tarea (este tipo de aplicación se explicará en el capítulo 5). Un segundo modo de utilización de la métrica sería para la toma de decisiones para llevar a cabo un proceso de defragmentación, y será explicado en detalle en el capítulo 7.

4.1 Métrica basada en el número y la complejidad de los huecos

Se puede disponer de una estimación precisa del estado de fragmentación de la FPGA en cualquier momento usando una métrica de fragmentación. Esta técnica de estimación se basa en el número de huecos en el área libre y en su complejidad, medida de forma simple mediante el número de vértices de cada hueco.

4.1.1 Presentación de la métrica

En esta métrica, el nivel de fragmentación del área libre para un estado de una FPGA podría estimarse como:

$$FH = 1 - \prod_i \left[\left(\frac{4}{V_i} \right)^n \times \frac{A_i}{A_{L\_FPGA}} \right] \qquad (4.1)$$

Donde el término entre corchetes representa cómo de "adecuado" es un hueco determinado, con un área A_i y V_i vértices para acoger nuevas tareas:

- $(4/V_i)^n$ representa la adecuación de la forma del hueco *i* para acomodar tareas rectangulares. Resaltar que cualquier hueco con cuatro vértices tiene la forma más adecuada. Para la mayoría de los experimentos se ha empleado n=1, pero se pueden emplear valores diferentes para n dependiendo de lo que se quiera penalizar la aparición de huecos con formas complejas y por tanto difíciles de utilizar.

- (A_i /A_{L_FPGA}) representa el área del hueco relativo normalizado. A_{L_FPGA}, representa el área libre total de la FPGA. Esto es $A_{L\_FPGA} = \sum A_i$.

La figura 4.1 muestra diferentes situaciones en una FPGA donde aparece el valor de fragmentación (métrica FH) dado por la métrica basada en la complejidad de huecos.

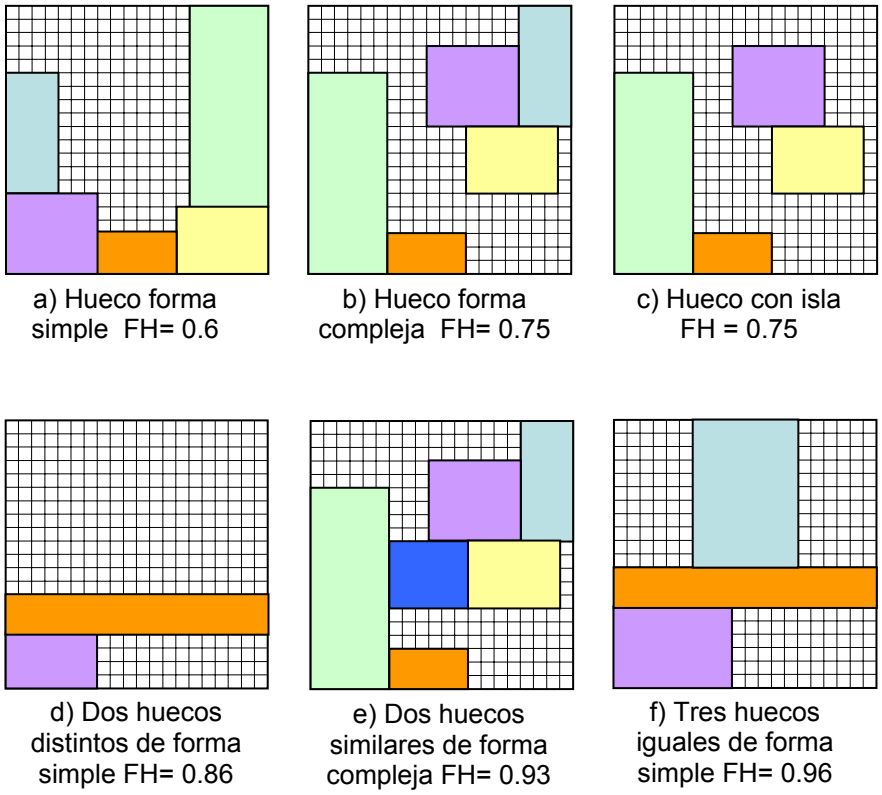


Figura 4.1. Diferentes situaciones en una FPGA y sus valores de fragmentación calculados con FH.

En la figura 4.1.a se muestra que todo el espacio libre se encuentra en un hueco con forma simple en el que se podría ubicar una tarea que ocuparía el 78% de la superficie libre. En esta situación la métrica da un valor de 0.6, cuando de manera intuitiva se espera un valor bajo de la fragmentación.

En la figura 4.1.b y 4.1.c, muestran dos situaciones distintas con dos valores de fragmentación iguales. Sin embargo, para el estado mostrado en (c) se debería esperar un valor mayor de fragmentación ya que la superficie total

libre es mayor y sin embargo el tamaño máximo de tarea que se podría insertar es el mismo. Además, en la parte (c) aparece una isla formada por dos tareas y la métrica FH no tiene en cuenta esta situación.

Si comparamos las situaciones mostradas en la parte inferior de la figura en (d) (e) y (f), comprobamos que para una situación de alta fragmentación del espacio libre pero con sólo dos huecos en (e), tenemos un valor de FH=0.93, mientras que en (f) el valor de FH es mayor porque el espacio libre está formado por tres huecos, aún cuando se podrían insertar tareas de mayor tamaño que en (e). Para la situación mostrada en (d) el valor de FH es menor, como se puede esperar intuitivamente porque sólo hay dos huecos con una forma muy "adecuada" y se podrían insertar tareas de mayor tamaño que en (e) y (f).

4.1.2 Valoración crítica de la Métrica

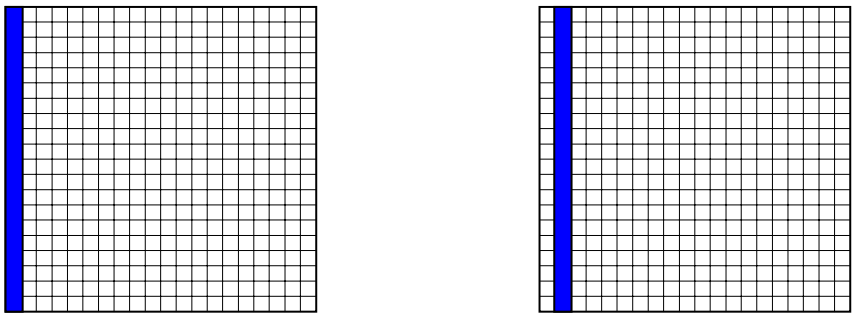
Después de realizar varios experimentos y aplicar la métrica en distintas situaciones como las anteriores, se ponen en evidencia ciertas limitaciones de la métrica FH, que aunque tienen importancia, se pueden solventar con la introducción de dos mejoras que se describen a continuación.

- *Mejora 1*. Después de analizar el comportamiento de la métrica propuesta en las distintas situaciones mostradas en la figura 4.1 parece interesante modular el valor de *n* para poder responder con valores esperados en las distintas situaciones de fragmentación mostradas en dicha figura.

Además, esta métrica de fragmentación penaliza excesivamente la creación de huecos sin tener en cuenta su tamaño relativo. En la figura 4.2 muestra una FPGA con una única tarea que ocupa una columna de la FPGA.

La métrica de fragmentación aplicada al estado mostrado en la figura 4.2.a, nos proporciona como se puede intuir un valor de cero. Sin embargo, en el estado 4.2.b, que es muy similar a la otra situación (a), da un valor próximo al máximo de 1. Por tanto, esta métrica tiende muy rápido a 1 si existe un hueco

separado de pequeña área, cuando debería mostrar un comportamiento más lineal y suave.



a) FH= 0 **b) FH= 0,95**

Figura 4.2. Situaciones similares en una FPGA y valores de fragmentación FH proporcionados por la métrica.

En la tabla 4.1 aparecen los valores que proporciona la métrica de fragmentación FH aplicada a los distintos escenarios mostrados en la figura 4.1 cuando se varía el valor de *n* de la fórmula (4.1).

Tabla 4.1. Valores de fragmentación FH para distintos escenarios de FPGA en función de *n*.

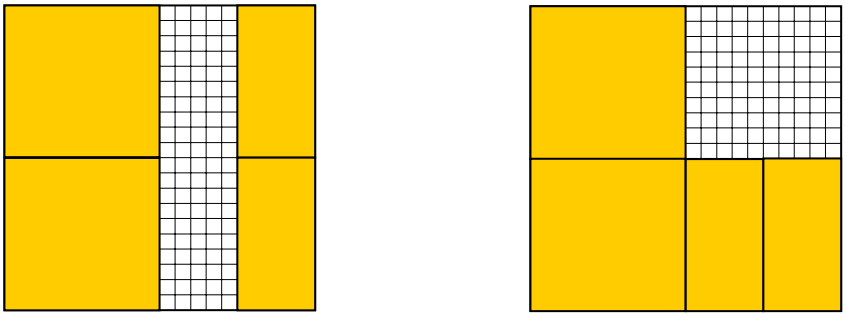
n	**a**	**b**	**c**	**d**	**e**	**f**
1/4	0,2	0,29	0,29	0,86	0,91	0,96
1/2	0,36	0,5	0,5	0,86	0,93	0,96
3/4	0,49	0,64	0,64	0,86	0,95	0,96
1	0,6	0,75	0,75	0,86	0,93	0,96
2	0,84	0,94	0,94	0,86	0,99	0,96

Como muestra la tabla 4.1, valores de $n \geq 3/4$ penalizan en exceso las formas complejas de los huecos, porque proporciona valores de FH demasiado elevados en el escenario (a), cuando se esperan valores de FH menores porque todo el espacio libre está fusionado en un solo hueco que tiene además un aspecto bastante cuadrado. Concretamente en la situación descrita en 4.1.a sería posible la ubicación de tareas de gran tamaño, que tuviesen hasta el 78% del tamaño de todo el espacio libre.

Por otra parte, cuando *n*=1/4, proporciona valores muy parecidos para las escenarios (a) y (b), cuando el escenario (b) tiene distribuido el espacio libre en un solo hueco, pero de forma muy irregular, por lo que se espera un valor sensiblemente mayor de FH. Los escenarios representados en (d) y (f) no varían cuando modulamos el valor de *n* porque tienen huecos de forma cuadrada y el valor de FH no está penalizado.

Concluyendo, el valor *n*=1/2 parece ser el más adecuado para proporcionar valores cercanos a los esperados en los distintos escenarios que aparecen en la figura 4.1.

Sin embargo, la métrica propuesta en (4.1) no penaliza la aparición de islas como sucede en el escenario (c) y tampoco favorece las formas de hueco con "aspecto más cuadrado", ya que considera igual de bueno cualquier hueco rectangular, sin favorecer los más cuadrados, con geometrías que intuitivamente parecen más favorables para acoger mayor número de tareas entrantes. En la figura 4.3 se muestran dos estados de una FPGA de 20*20 BBRs, con las mismas tareas en ejecución pero distribuidas de modo que en (a) el espacio libre resulta menos aprovechable que en (b) donde presenta un aspecto más cuadrado y previsiblemente se podrá ubicar una gama más amplia de tareas. Cuando se estima el nivel de fragmentación con la métrica FH proporciona el mismo valor para los dos estados.



a) FH= 0 **b) FH= 0**

Figura 4.3. Estados diferentes en una FPGA con valores de fragmentación FH iguales.

- *Mejora 2*. Como se puede apreciar, una FPGA es globalmente menos adecuada si contiene más huecos separados de área sin ocupar. El valor de

FH aumenta rápidamente con el número de huecos. Sin embargo, debe tenerse en cuenta que sólo se deberían considerar en este cálculo aquellos huecos que encierran una cantidad significativa de área libre de la FPGA. En esta aproximación se ha decidido empíricamente que para que un hueco A_i se tenga en cuenta se debe cumplir $A_i \geq A_{L\_FPGA}/10$. Como se puede comprobar, con esta simple mejora, el valor de fragmentación en situaciones como el mostrado de la figura 4.2.b, se modifica y no tiende rápidamente a 1.

4.2 Métrica basada en la cuadratura del perímetro

Para suplir las carencias de la métrica FH basada en la complejidad del hueco, se ha ideado una segunda métrica con el principal objetivo de separar la estimación de la fragmentación del grado de ocupación y reducir a la vez el impacto del número de huecos.

4.2.1 Caso de un único hueco

Esta nueva métrica parte de una idea muy simple: se considera un hueco libre ideal, aquel que es capaz de acomodar una mayoría de las tareas que llegan con una variedad de formas y un área de tarea total, similar o menor que el tamaño del hueco. Supondremos que el hueco libre ideal debería tener una forma cuadrada perfecta, de modo que ese hueco sería el que es capaz de acomodar a una mayoría de tareas entrantes. Una de las ventajas adicionales de una forma cuadrada sería que las interconexiones de longitud máxima dentro de la tarea serían más cortas que para tareas con formas rectangulares con el mismo tamaño de área, o incluso con formas irregulares.

Partiendo de las consideraciones anteriores, se define la nueva métrica de fragmentación del área libre para un estado de una FPGA como:

$$FQ = 1 - Q \tag{4.2}$$

Donde el término *Q* define para cualquier hueco *H*, con un área *A* y cualquier forma arbitraria, su **cuadratura** relativa, que significa "cuánto se aproxima su forma a la del hueco ideal consistente en un cuadrado perfecto". La magnitud *Q* de la cuadratura relativa se estima dividiendo el área real *A* del hueco *H* entre el área A_Q de un cuadrado perfecto con el mismo perímetro *P* que dicho hueco, donde A_Q se calcula como:

$$A_Q = \left(\frac{P}{4}\right)^2 \tag{4.3}$$

y por tanto la cuadratura relativa *Q* es:

$$Q = \frac{A}{A_Q} \tag{4.4}$$

Como se puede comprobar, la métrica basada en la cuadratura FQ, definida en (4.2), considerará que la fragmentación de un hueco dado *H* es mínima cuando tenga una forma cuadrada. Por el contrario, cuanto más largo sea el perímetro para el mismo tamaño de área, se obtendrá un valor más alto de fragmentación.

En la figura 4.4 se muestra una FPGA de 20x20 BBRs con un conjunto de tareas en ejecución, situadas en diferentes posiciones y que proporcionan distintos estados de fragmentación del área libre. El área libre en todos los estados es A=169 unidades de BBR, pero el perímetro P y por tanto los valores A_Q y *Q* son diferentes para cada uno de ellos. Además en la parte izquierda se muestra un hueco de igual tamaño, pero organizado de forma óptima, por lo que *Q* = 1 y *FQ* = 0. De ese modo la fragmentación FQ es distinta en cada uno de los estados mostrados desde (a) hasta (f), y es menor para la FPGA cuya forma del área libre es más apta para acomodar las tareas que en el futuro van llegando, como se muestra en 4.4.f.

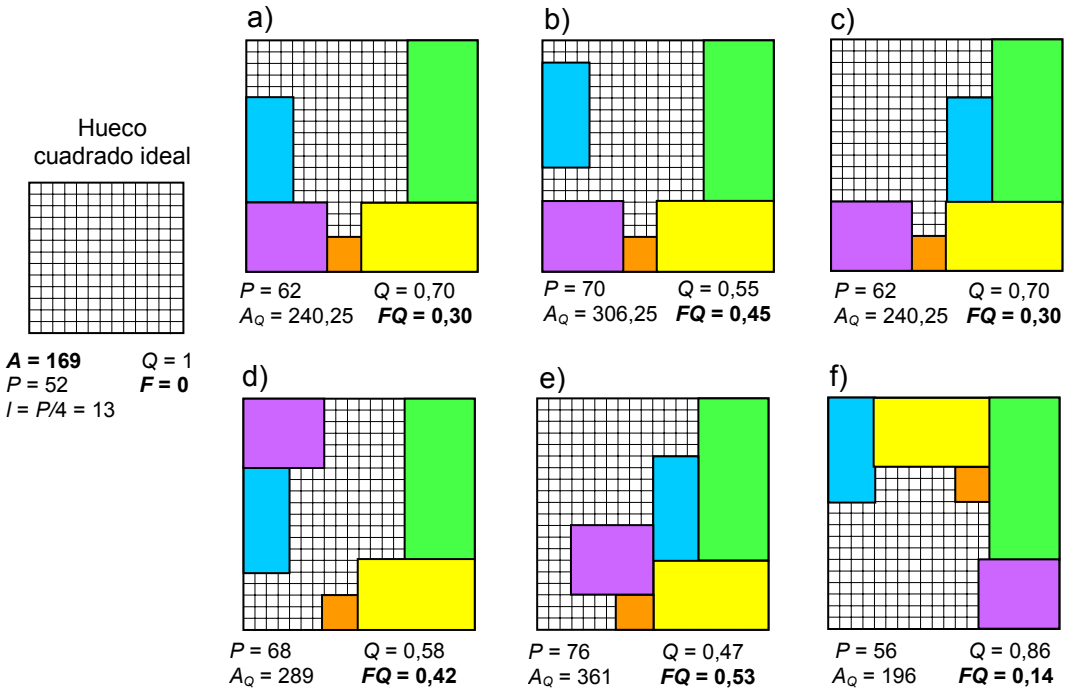


Figura 4.4. Valores de la métrica FQ para diferentes huecos con el mismo área pero distinta geometría.

4.2.2 Caso de múltiples huecos

Esta métrica y el concepto de cuadratura relativa se pueden extender directamente a un área libre formado por varios huecos, considerando la frontera entre el área libre y ocupada como un único perímetro y calculando su perímetro total como la suma de los perímetros de todos los huecos del CLV, cada uno definido por su LV.

Se define no una cuadratura para cada uno de los huecos sino una cuadratura global *Q*, donde los valores del perímetro total *P* y del área total *A* de todos los huecos serían:

$$P = \sum_i P_i \quad \text{y} \quad A = \sum_i A_i \tag{4.5}$$

Y aplicamos (4.2) tal cual con los valores de A_Q y *Q* calculados como en (4.3) y (4.4).

El valor de fragmentación sería entonces una medida de cuánto dista el espacio libre disponible total delimitado por *P*, de ser un hueco único ideal.

La figura 4.5 muestra varias situaciones para la misma FPGA de 20x20 BBRs y las cinco tareas en ejecución que se mostraron en la figura anterior. Ahora las cinco tareas están situadas de un modo diferente, y el área libre A está dividido en dos (en (a) y (b)), o incluso en tres huecos independientes (en (c)).

En dicha figura podemos observar que esta nueva métrica no necesita tener en cuenta el número de huecos para estimar el nivel de fragmentación en estados de ocupación complejos ya que maneja el área libre globalmente, simplificando por tanto el cálculo. También se puede observar que, al contrario de lo que ocurría con la métrica FH, cuando aumenta el número de huecos no se dispara la fragmentación si estos huecos tienen formas simples.

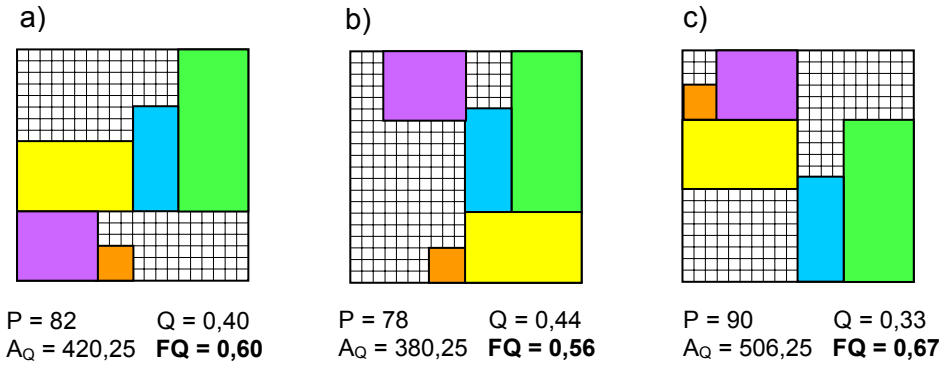


Figura 4.5. Valores de la métrica para diferentes ubicaciones de tareas y para múltiples huecos.

4.2.3 Caso de islas

Una situación que la métrica FQ trata automáticamente es la aparición de islas, que ya fue descrita en la sección 3.4.2. Las islas son situaciones de alta fragmentación, no deseables, que pueden aparecer cuando algunas tareas finalizan y salen de la FPGA, mientras otras tareas continúan su ejecución. Es importante que una métrica de fragmentación sea capaz de poder identificar y valorar estas situaciones especiales.

Esta nueva métrica FQ las trata automáticamente porque en la representación del perímetro del área libre (Lista de Vértices), la isla está conectada al resto del perímetro a través de aristas virtuales, como se representa en la figura 4.6. Estas aristas virtuales son consideradas como parte del perímetro cuando se calcula *P* como en (4.5). De ese modo, una isla cercana al perímetro tendrá unas aristas virtuales cortas y el valor de *P* será menor que en otras situaciones donde la isla esté más alejada.

Una isla, incluso una de pequeño tamaño, puede ser bastante molesta cuando está localizada en el centro de un hueco grande. Por este motivo se ha añadido a estas aristas virtuales un factor de peso asociado configurable que multiplique su longitud cuando se desee penalizar la aparición de dichos eventos.

La figura 4.6 muestra cómo la nueva métrica FQ tiene en cuenta la distancia de la isla respecto del perímetro del hueco. La figura muestra tres situaciones similares, con sus correspondientes LVs en la parte superior, donde se aprecia una isla unida a la LV a través de sus correspondientes aristas virtuales, en línea punteada.

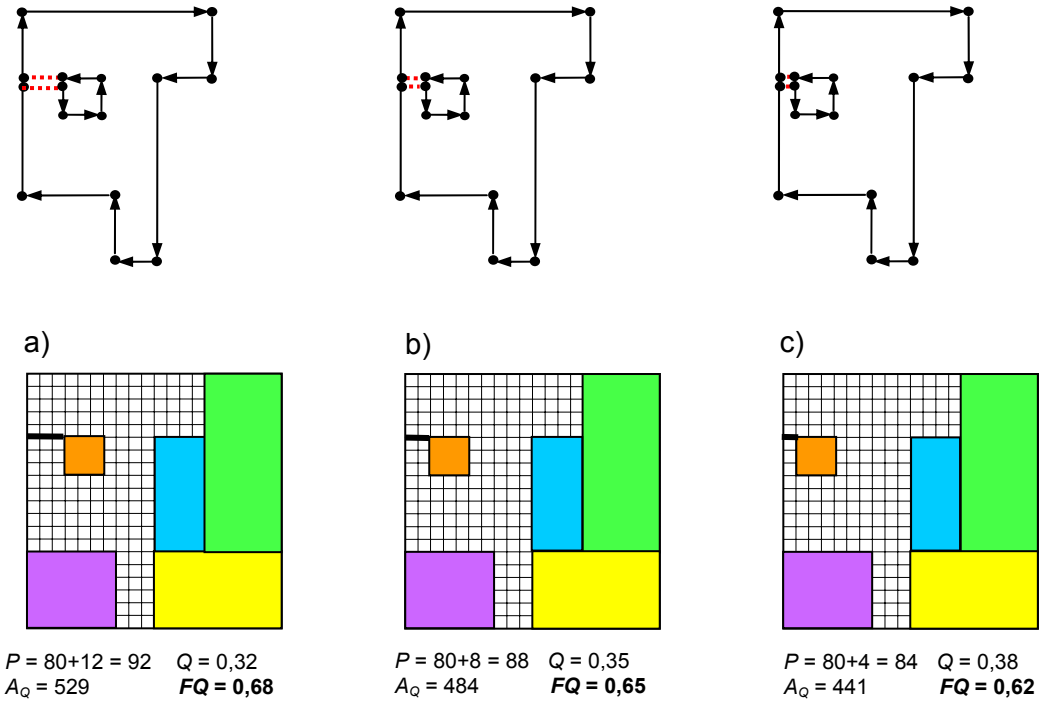


Figura 4.6. Valores de la métrica para un hueco con una isla en diferentes ubicaciones.

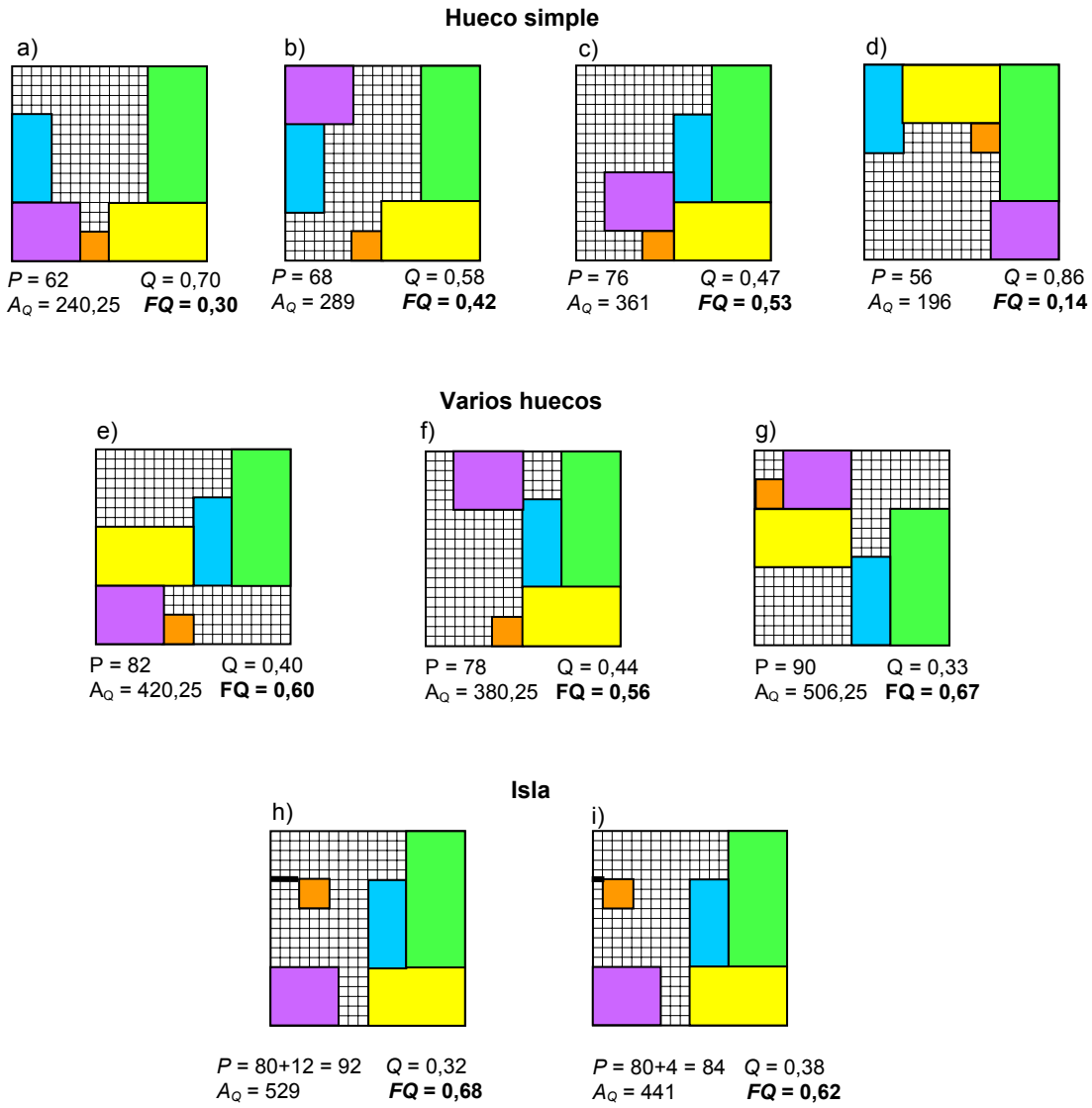
El valor de fragmentación más alto corresponde a la figura 4.6.a mientras que para las figuras 4.6.b y 4.6.c es menor porque la isla está más cerca del perímetro. En este ejemplo se han ponderado las aristas virtuales con factor de penalización de 2.

Esta métrica es muy fácil de calcular, al menos para un algoritmo de *Asignación* (de recursos a tareas) que tenga control y acceso a la frontera del área libre descrita por la Lista de Vértices, ya que el perímetro es una medida directa de la longitud de la LV.

4.3 Comparación de métricas de fragmentación

Para comparar esta nueva métrica con la propuesta en la sección 4.1, o con las descritas en la sección 2.2 de otros grupos de investigación, se han calculado los valores de fragmentación proporcionados por algunas de esas métricas en algunos de los ejemplos simples de FPGAs mostrados en las figuras 4.4, 4.5 y 4.6 que corresponden a estados de una FPGA con un hueco simple, varios huecos y con una isla, respectivamente. Estos resultados se muestran en la tabla de la parte inferior de la figura 4.7 donde también aparece el tamaño del rectángulo libre MER mayor disponible (MAX_MER), que aunque no es viable como una técnica real debido a su alta complejidad, si que puede ser usado como una referencia. En la parte superior de la figura se reproducen de nuevo los ejemplos seleccionados de las figuras 4.4, 4.5 y 4.6 para facilitar el seguimiento de los resultados.

El propósito de esta tabla es mostrar que el valor de fragmentación calculado por la nueva métrica (referenciada como FQ, con el valor de cuadratura *Q* entre paréntesis) es una estimación fiable y ajustada del estado de fragmentación de una FPGA.



	Hueco simple				Varios huecos			Isla	
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**
F1 (Wigley)	10	7	10	**11**	7	10	7	**6**	6
F2 (Walder)	0.16	0.36	0.32	**0.14**	0.39	0.32	0.39	**0.44**	0.45
F3 (Handa)	0.07	0.18	0.21	**0.08**	0.28	0.22	0.33	**0.21**	0.20
F4 (Ejnoui)	0.58	0.58	0.58	**0.58**	0.96	0.98	1	**0.58**	0.58
FH	0.60	0.67	0.60	**0.60**	0.89	0.95	0.99	**0.67**	0.67
FQ	**0,30**	**0,42**	**0,53**	**0,14**	**0,60**	**0,56**	**0,67**	**0,68**	**0,62**
(Cuadratura)	**(0,70)**	**(0,58)**	**(0,47)**	**(0,86)**	**(0,40)**	**(0,44)**	**(0,33)**	**(0,32)**	**(0,38)**
MAX_MER	140	98	110	**143**	80	110	80	**70**	84

Figura 4.7. Comparativa entre diferentes métricas de fragmentación.

Cuando se compara FQ con los valores de MAX_MER, los casos de fragmentación mayor y menor se corresponden, al igual que sucede con la mayoría de los casos restantes. Solamente para los casos (b) y (c) hay una diferencia apreciable, que es debido al hecho de que en el caso (b) existen muchos rectángulos libres MER de tamaño medio, todos ellos adecuados para acomodar futuras tareas, aunque el mayor MER sea menor que en otros casos con valores mayores de FQ.

Para el resto de métricas, se observa que F1, F2 y F3 no tienen un buen comportamiento cuando aparecen islas porque no discriminan entre las situaciones (h) y (i) (F2 incluso proporciona un valor mayor de fragmentación en el caso (i), cuando la isla está más cerca del perímetro). Además F3 considera menos fragmentado el caso (a) respecto de (d) cuando el último tiene un tamaño mayor de MAX_MER y de aspecto más cuadrado. Finalmente, F4 y FH no discriminan entre muchos de los casos propuestos y asignan valores de fragmentación excesivos en los casos donde hay varios huecos separados como en (e), (f) y (g).

La figura 4.8 muestra una representación gráfica meramente cualitativa, donde se ha normalizado cada métrica para obtener valores visualizables en una escala común. La gráfica ilustra claramente esta afinidad entre FQ y MAX_MER.

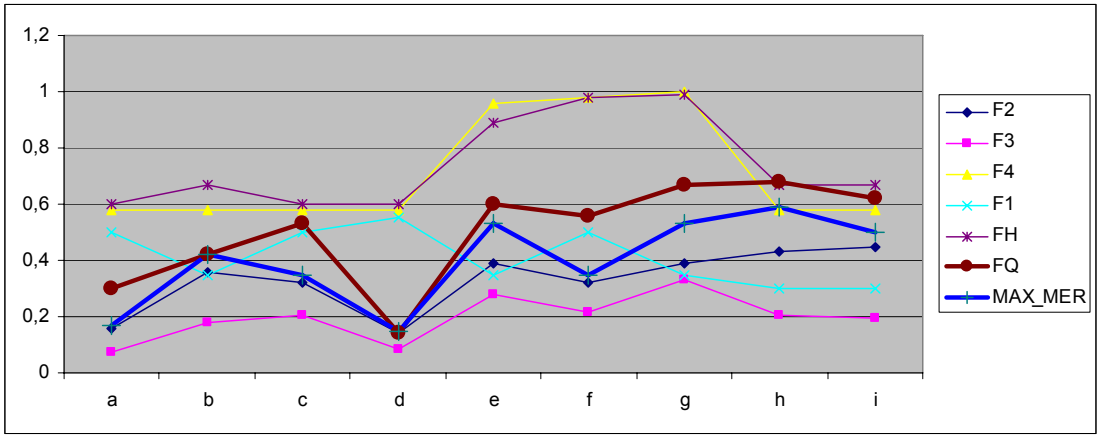


Figura 4.8. Comparativa entre diferentes métricas de fragmentación

4.4 Conclusiones

La sección anterior 4.3 ha mostrado cómo la nueva métrica FQ es la que es capaz de asignar valores de fragmentación más apropiados para cada situación de la FPGA.

Hay que resaltar que la métrica FQ es mucho más simple de calcular que la otra métrica propuesta FH, porque no es necesario considerar cada hueco independiente del área libre de la FPGA. Cuando estas métricas son utilizadas por un *Gestor de HW* que utiliza la Lista de Vértices, como el propuesto en este trabajo de investigación, el perímetro del área libre es exactamente la longitud de la lista de vértices, por lo que la métrica FQ se adapta al *Gestor de HW* de forma natural. Además su cálculo es mucho más simple que el resto de las métricas y además no necesita de estructuras de datos complejas adicionales y específicas para el cálculo de la fragmentación, con un alto coste de actualización.

Capítulo 5:

Heurísticas de selección de vértices basadas sólo en el área (2D)

Una vez que ya han sido explicados en el capítulo 3 el algoritmo de gestión básico y la estructura CLV, se debe afrontar el problema de seleccionar un vértice de entre todos los candidatos viables que permitan la ubicación de una tarea dada. El módulo *Selector de vértice* realiza este proceso. Para hacer esta elección se necesitan por una parte un conjunto de posibles posiciones en las que ubicar la tarea que está demandando recursos HW para su ejecución, y por otra un criterio por el cual seleccionaremos una posición concreta frente a un subconjunto de todas las posiciones viables para realizar la ubicación de la tarea.

Para agilizar el proceso de búsqueda de la posición, será determinante el tipo de representación del espacio libre que se haya seleccionado, que podrá contener todas las posibles ubicaciones. Tendremos una situación muy diferente, por ejemplo, si se emplea un array bidimensional (en el que cada elemento se corresponda con un BBR, que podrá estar marcado como libre/ocupado, según su disponibilidad) con todas las posibles posiciones dentro de la FPGA, o que si usamos un subconjunto de posiciones, seleccionado con algún tipo de heurística. El criterio utilizado para seleccionar la ubicación final de la tarea de entre todos los posibles candidatos, será de vital importancia para optimizar la utilización de la FPGA y para que en el futuro la probabilidad de encontrar espacio para las tareas sucesivas sea mayor.

En primer lugar, se puede utilizar una solución simple basada en un criterio First Fit (FF), como se puede encontrar en [TSMM03], que tiene como única virtud que no precisa de búsqueda. Sin embargo, una alternativa más interesante es una solución Best Fit (BF) que intente encontrar la mejor ubicación de acuerdo a una función de coste dada. Cuando se aplica una solución BF sobre el CLV, cada LV_i se recorre completamente y se computa y almacena un valor para cada vértice candidato, de acuerdo a la función de coste de la heurística seleccionada. Finalmente la tarea se aloja en el vértice candidato más adecuado según el criterio de ubicación escogido. Se han desarrollado diferentes heurísticas que serán utilizadas como función de coste, que serán mostradas a continuación en este capítulo y en el siguiente.

En concreto, en este capítulo se describen dos heurísticas BF que, aunque realizan sus estimaciones basándose en el área, están apoyadas en dos principios diferentes: fragmentación y adyacencia.

5.1 Heurísticas basadas en Fragmentación

La posibilidad de utilizar las métricas de fragmentación como función de coste para la asignación de unos recursos concretos de la FPGA a una tarea ya fue adelantada en el capítulo 1. La idea trata de aprovechar las

características de las métricas de fragmentación, en concreto su habilidad para detectar cuándo el área libre de la FPGA es más o menos apta para acomodar las futuras tareas que lleguen, es decir, tiene que detectar si ese área está organizada eficientemente o ineficientemente, y proporcionar un valor numérico para esa organización. Cuando se considera un vértice candidato en concreto se supone la tarea ubicada en esa posición y se estima el nivel de fragmentación futura del hueco, o huecos resultantes. Esta medida nos indicará la probabilidad de acomodar la mayoría de tareas que lleguen en el futuro con una variedad de formas y un área de tarea similar o menor que el tamaño del hueco *H.*

Las heurísticas BF_FH y BF_FQ, usando como funciones de coste las métricas FH y FQ respectivamente, pueden estimar para cada vértice candidato viable V_i, la fragmentación producida en el área libre de la FPGA después de suponer ubicada la tarea en dicho vértice. El *Selector de vértice* considera cada una de las alternativas y finalmente ubica la tarea entrante $\boldsymbol{T_N}$ en aquella posición donde se genera el menor nivel de fragmentación.

La heurística BF_FH, basada en la métrica FH, calcula como función de coste para cada vértice candidato V_i:

$$FH_i = FH\ (CLV+T_N\ \ ubicada\ en\ V_i) \tag{5.1}$$

Mientras que la BF_FQ, basada en la métrica FQ, realiza como cálculo:

$$FQ_i = FQ\ (CLV+T_N\ \ ubicada\ en\ V_i) \tag{5.2}$$

En cada caso, la heurística elije como vértice para ubicar la tarea el que proporciona un menor valor de FH_i ó FQ_i .

La figura 5.1 muestra un ejemplo de funcionamiento de estas heurísticas, con el estado inicial de la FPGA y la LV correspondiente, situados en la parte superior. Para cada una de las situaciones se muestra el valor de las dos funciones de coste anteriores, FH basada en la complejidad de los Huecos y FQ basada en la cuadratura del perímetro. Cuando llega una nueva tarea $\boldsymbol{T_N}$ de 4*5 BBRs, se calcula el nivel de fragmentación para todas las posibles ubicaciones de la tarea que correspondan con vértices candidatos.

Con cualquiera de las dos heurísticas, los candidatos V1-V7 y V4-V5 producirán el menor nivel de fragmentación en el área libre resultante, por lo tanto uno de ellos será elegido por estas heurísticas para la ubicación de la tarea.

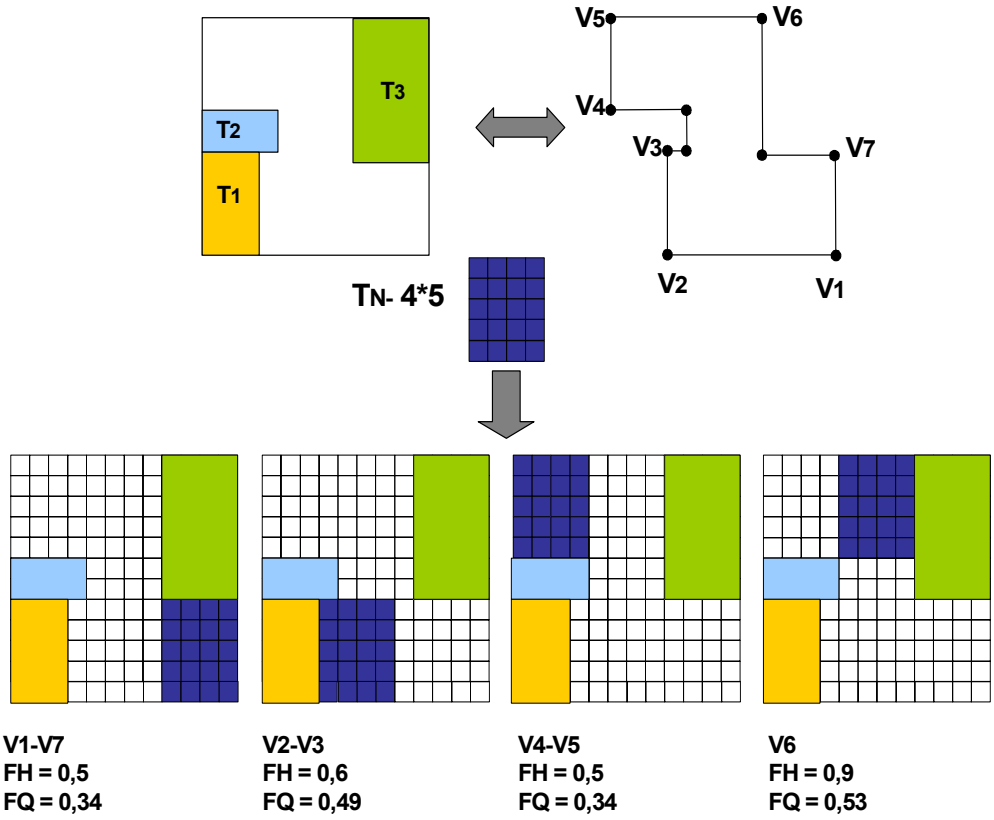


Figura 5.1. Ejemplo de heurísticas basadas en fragmentación.

Se han realizado experimentos comparando las heurísticas BF_FH y BF_FQ entre sí, y también con otras dos heurísticas basadas en MER que se han usado como referencia: una de tipo Best Fit, etiquetado como BF_MER (se elige el MER más pequeño capaz de alojar la tarea) y la otra de Worst Fit, WF_MER (se elige el MER de mayor tamaño) como los propuestos en [BaKS00].

No se han utilizado como función de coste otras métricas de fragmentación, como las expuestas en la sección 4.3, debido a la dificultad de su

programación y de su integración en el entorno del Gestor de HW, y que en algunos casos no era ni siquiera posible.

Los resultados experimentales aparecen resumidos en la tabla 5.1 y en las figuras 5.2 y 5.3. Se ha utilizado una FPGA de 20*20 BBRs, y como banco de pruebas distintos lotes de 100 tareas, cada uno con características diferentes.

Los lotes se componen de cuatro rangos de tamaño de tareas diferentes. El lote S1 esta compuesto por tareas pequeñas, con unas dimensiones de lx_i y ly_i generadas aleatoriamente y que varían de 1 a 10 unidades. El lote S2 esta compuesto de tareas de tamaño mediano, con unas dimensiones que varían de 2 a 14 unidades. El lote S3 está formado por tareas de gran tamaño, con unas dimensiones de lados que varían desde 4 a 18 unidades. S4 es el lote más heterogéneo, con tareas pequeñas, medianas y grandes combinadas. El número medio de tareas en ejecución depende del tamaño de tarea medio y es aproximadamente 12 para S1, 8 para S2, 6 para S3 y para S4 es más impredecible.

Tabla 5.1. Resumen de resultados experimentales.

	Parámetro	S1-F	S1-D	S1-N	S2-F	S2-D	S2-N	S3-F	S3-D	S3-N	S4-F	S4-D	S4-N
WF_MER	Nº. ciclos	144	158	158	146	186	199	154	256	302	152	212	200
	% área	67	68	68	66	68	66	72	72	73	66	62	68
	Vol. rech.	12	0	0	27	4	0	50	17	0	25	5	0
BF_MER	Nº. ciclos	147	156	156	141	192	203	154	264	321	152	197	207
	% área	69	69	69	65	67	65	72	71	68	63	68	66
	Vol. rech.	7	0	0	31	3	0	50	16	0	29	3	0
BF_FH	Nº. ciclos	145	154	154	141	181	188	153	265	294	156	207	196
	% área	68	70	70	68	71	70	72	72	75	65	64	70
	Vol. rech.	9	0	0	28	3	0	50	14	0	28	3	0
BF_FQ	Nº. ciclos	143	150	150	144	180	190	150	265	300	148	194	194
	% área	71	72	72	72	71	70	75	73	73	66	70	70
	Vol. rech.	7	0	0	22	3	0	49	12	0	27	0	0

Todos los lotes de tareas tienen un exceso de carga de trabajo que fuerza al *Planificador de tarea* a almacenar algunas tareas temporalmente en la cola de espera C_{esp}, y en algunos casos incluso a rechazarlas cuando no se cumple la restricción temporal de su tiempo límite de arranque t_lim_i, dado por la condición (3.2).

Para cada uno de los lotes, se han usado tres tipos de restricciones temporales: fuerte (F), débil (D) o no existente (N), de modo que los doce lotes de experimentos se han etiquetado como S1-F, S1-D, S1-N, S2-F, ... hasta S4-N.

Los parámetros usados para caracterizar este experimento son:

1. El volumen total de cálculo rechazado por los algoritmos de gestión (Vol. rech), para cada lote de tareas, que es el sumatorio del volumen de cálculo de todas las tareas rechazadas, calculado como en (3.4). Este volumen representa todas las tareas que fueron rechazadas porque el Gestor de HW no fue capaz de encontrar una posición viable en tiempo para cumplir con los requisitos temporales de la tarea. Para tareas con la misma prioridad, este volumen representa, mejor que solamente el número de tareas rechazadas, la cantidad de carga de trabajo que el sistema no ha podido realizar.
2. El número de ciclos necesarios para completar el volumen total de cálculo. Es necesario resaltar que el número de ciclos solamente es significativo cuando no se han rechazado tareas y por tanto es un parámetro que permite una comparación directa entre las heurísticas.
3. El nivel de ocupación promedio de la FPGA (% área) obtenido para cada lote de tareas. Este parámetro representa la fracción de la capacidad de procesamiento del total de la FPGA, que el algoritmo de gestión ha sido capaz de explotar para un conjunto dado de tareas. Para considerar el efecto de las tareas rechazadas, hemos “modulado” la ocupación media de la FPGA, multiplicándola por el porcentaje de volumen de cálculo ejecutado de manera satisfactoria.

Los resultados de la tabla 5.1 se muestran en las siguientes figuras. La figura 5.2 muestra la cantidad de volumen de cálculo rechazado para cada

lote de tareas por cada una de las heurísticas de selección, con restricciones temporales fuertes (a) y débiles (b). El resto de las tareas se supone que han sido cargadas y ejecutadas de manera satisfactoria antes de dejar de cumplir la condición (3.2).

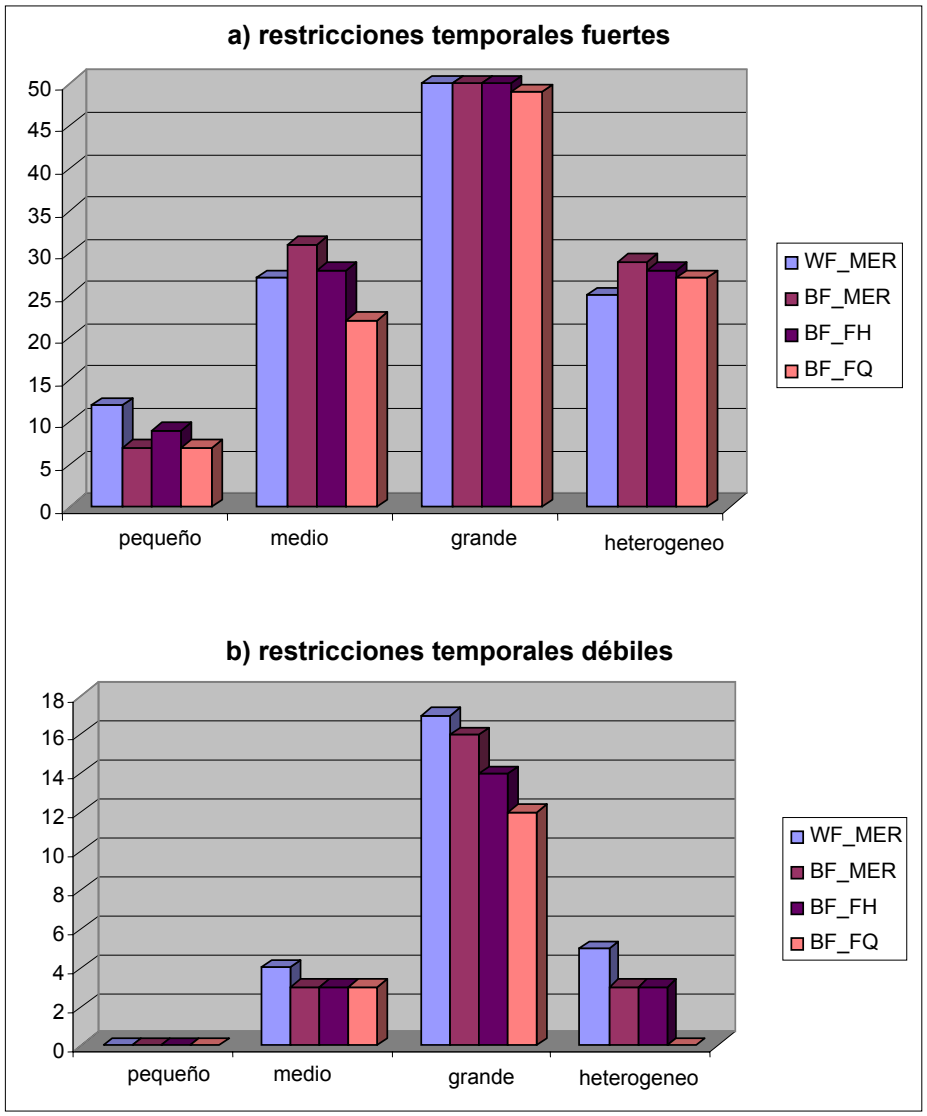


Figura 5.2. Porcentaje de volumen de cálculo rechazado, para lotes de tareas con restricciones temporales fuertes (a) y débiles (b).

Se puede observar en la figura 5.2 que la heurística BF_FQ da resultados similares o mejores al resto en todos los casos.

Cuando no hay restricciones temporales, ninguna tarea es rechazada y la comparación se establece en términos de la cantidad de ciclos necesarios

para completar la ejecución de todo el lote de tareas. La figura 5.3 muestra que las heurísticas basadas en fragmentación BF_FQ y BF_FH son capaces de ejecutar toda la carga de trabajo de cada lote de tareas en menos ciclos que las basadas en rectángulos MER, para la mayoría de lotes de tareas.

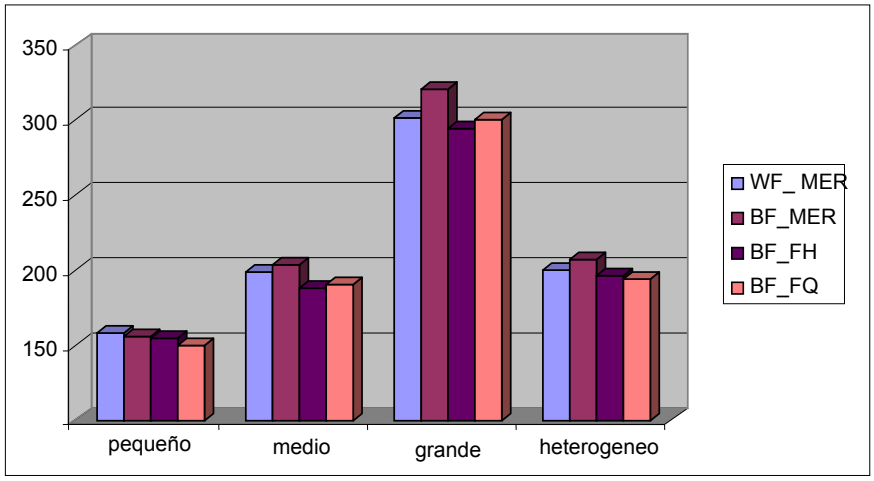


Figura 5.3. Número de ciclos necesarios para lotes de tareas sin restricciones temporales.

La figura 5.4 muestra como la ocupación media en la FPGA se comporta de un modo similar.

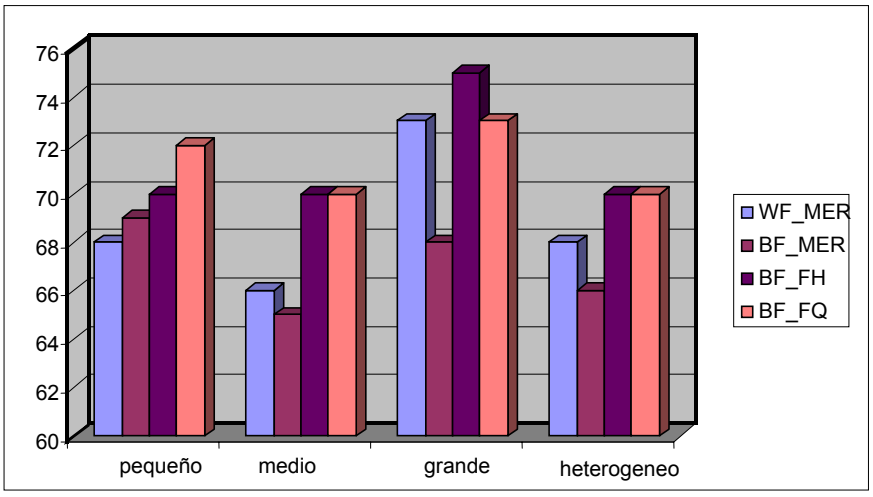


Figura 5.4. Ocupación media del área para lotes de tareas sin restricciones temporales.

El volumen de ocupación medio en la FPGA varia entre un 66 y un 75%, por lo que es de resaltar que una cantidad significativa de la FPGA (34 al 25%) no se puede utilizar debido a la fragmentación.

Es necesario notar que aunque las soluciones basadas en MER se dan sólo como referencia, porque su complejidad hace que no sea viable su uso en un entorno real de asignación *online*, sí que proporcionan una pista de cómo se comportarán otras heurísticas basadas en rectángulos (como las presentadas en el capítulo 2). Como las heurísticas de asignación de recursos HW que utilizan las métricas de fragmentación propuestas FQ y FH, se comparan de manera favorable con aquellas basadas en MER, podemos esperar que mejoren los resultados respecto de otras técnicas no-óptimas basadas en rectángulos no solapados NO-MER.

Aunque la heurística BF_FQ suele proporcionar resultados algo mejores que la BF_FH, no en todos los casos es así, y la diferencia no es siempre significativa. Sin embargo, como además la métrica FQ es mucho más simple de calcular que la FH, será la elegida en adelante como función de coste para ser comparada con el resto de heurísticas propuestas.

5.2 Heurística basada en adyacencia espacial 2D

El segundo tipo de heurística aplica una técnica muy intuitiva: coloca las tareas en aquellas posiciones donde se obtenga mayor grado de contacto entre las aristas de la tarea y las de la envolvente del área libre. El vértice seleccionado dependerá por tanto de la forma de la tarea y de la envolvente. Esta idea se muestra en la figura 5.5 donde se dispone de una FPGA en la que aparecen dos tareas en ejecución. El objetivo sería ubicar la nueva tarea en la posición donde quede el espacio libre resultante con una forma y proporción adecuadas para alojar tareas con unas dimensiones máximas en el futuro. Como se puede comprobar en la figura 5.5 en la posición 2, el espacio libre conseguiría alojar un rectángulo de dimensiones máximas. Para esta posición se puede comprobar que se cumple la condición de obtener el

máximo grado de contacto entre las aristas de la nueva tarea y de la envolvente del espacio libre. Cuando se intenta ubicar una nueva tarea, en una posición óptima según esta idea, se probaría en todas las posiciones esquina que coinciden y están definidas por vértices candidatos de la LV.

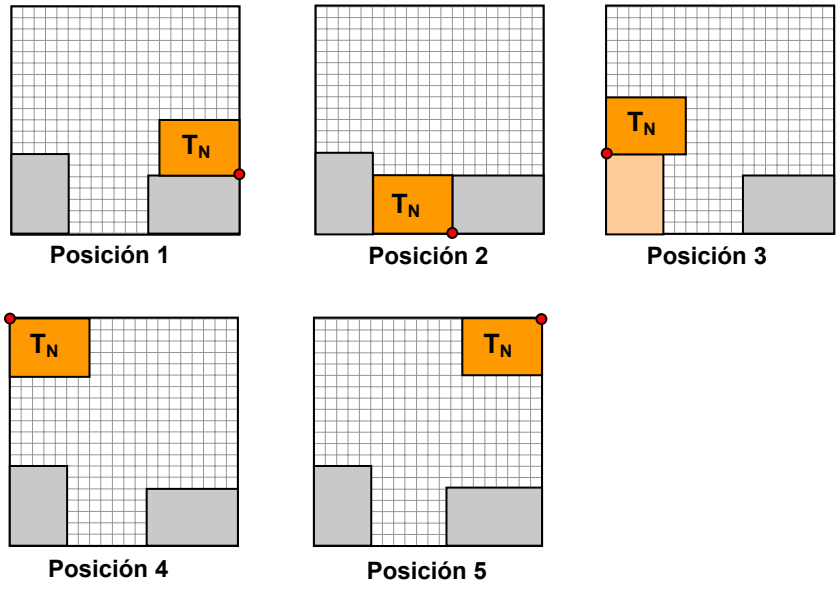


Figura 5.5. Ejemplo de heurística basada en Adyacencia 2D.

La heurística basada en la adyacencia 2D ubica la tarea T_N en la posición del vértice donde la tarea entrante consigue el máximo nivel de contacto entre las aristas de la tarea y las aristas del perímetro del área libre definido por el CLV. Este valor de adyacencia se calcula en términos de longitud de unidades de bloques básicos BBRs.

La adyacencia 2D calculada para cada vértice candidato V_i, es igual al sumatorio de la longitud del perímetro (frontera) de la nueva tarea T_N que está en contacto con cada arista *h* de la LV_j de cada hueco que forma el CLV, medido en BBRs:

$$Ad-2D_i = \sum_j \sum_h \left(LV\_arista_{jh} \cap T_N\_frontera \right) \qquad (5.3)$$

$$\text{con } \mathrm{LV_j} \in \mathrm{CLV} \text{ y } \mathrm{h} \in \mathrm{LV_j}$$

La figura 5.6 muestra un ejemplo simple para ilustrar el funcionamiento de esta heurística. El estado de la FPGA se muestra en la parte de arriba, con

dos tareas en ejecución y en la parte superior derecha se muestra la LV que define el espacio libre resultante.

Cuando una nueva tarea de 4*5 BBRs llega, esta solución calcula el valor de adyacencia 2D para cada posición candidata que sea viable y que estará indicada por los vértices correspondientes. Por lo tanto el algoritmo BF que usa el criterio de adyacencia 2D situaría la tarea en el quinto o sexto candidato (es la misma posición), donde el valor de adyacencia es máximo e igual a 12.

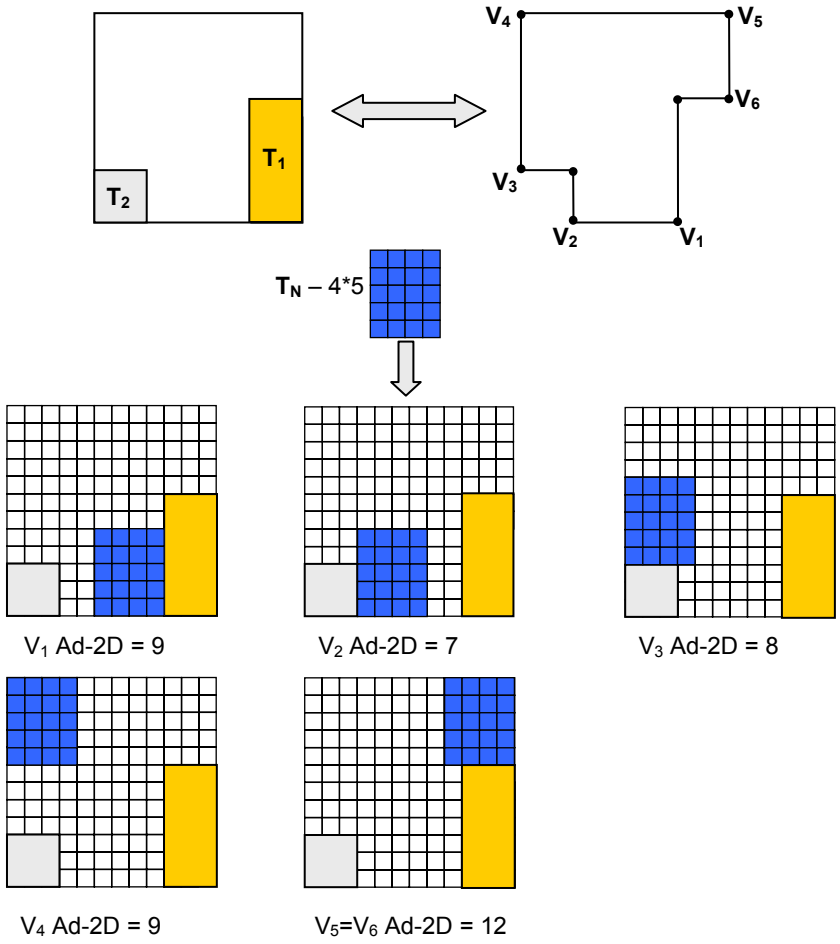


Figura 5.6. Ejemplo de heurística basada en Adyacencia 2D.

5.3 Resultados experimentales en 2D

Para evaluar la calidad de las distintas alternativas, se han realizado experimentos usando tres heurísticas de selección de ubicación diferentes:

- Lista de Vértices con FF (FF_LV).
- Lista de Vértices con BF y heurística de Adyacencia espacial (BF_Ad-2D).
- Lista de Vértices con BF y heurística de Fragmentación basada en Cuadratura (BF_FQ).

En la figura 5.7 se muestra un ejemplo de cómo estas diferentes heurísticas procesan una nueva tarea seleccionando diferentes ubicaciones viables y se comparan de un modo cuantitativo.

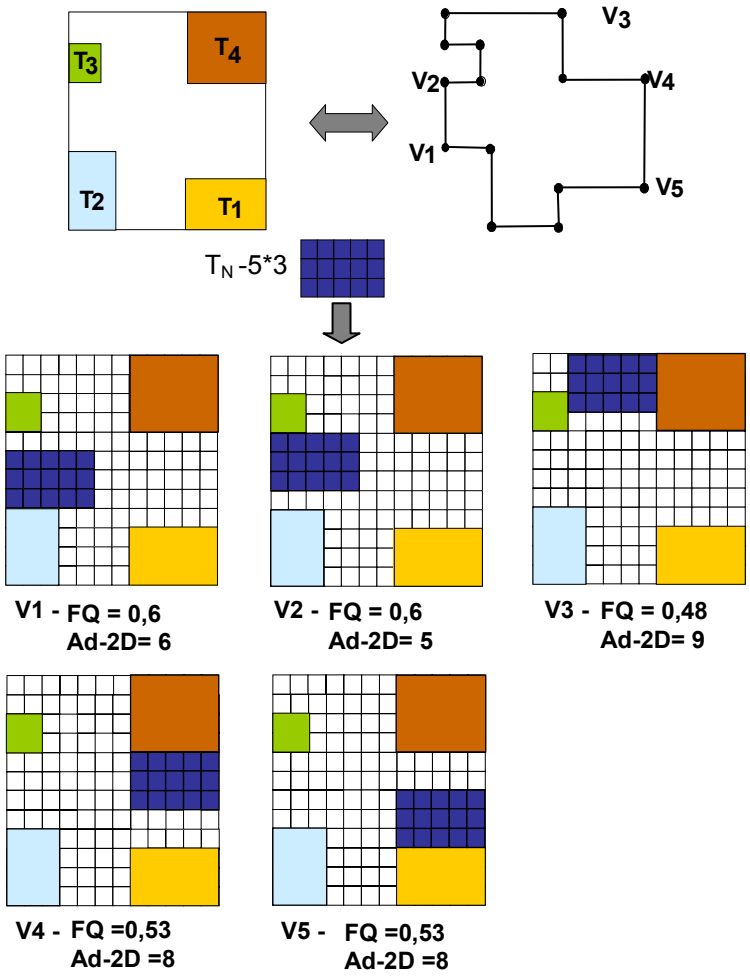


Figura 5.7. Valores de fragmentación y adyacencia para los diferentes vértices candidato.

La heurística FF_LV selecciona el vértice V_1 porque es el primero donde la tarea entrante cabe y es viable su ubicación, la fragmentación resultante estimada con FQ será de 0,6; las heurísticas BF_FQ y BF_Ad-2D seleccionarán el vértice V_3 donde la adyacencia es máxima e igual a 9, y obtiene el valor menor de fragmentación de FQ= 0,48. Los otros vértices candidatos tienen valores intermedios de fragmentación y adyacencia. Se puede observar que la heurística BF_Ad-2D, aunque es más simple de calcular, da resultados que coinciden con los obtenidos con BF_FQ, que utiliza la métrica de fragmentación.

Para poder comparar el funcionamiento de las dos métricas de manera cuantitativa, se ha utilizado una FPGA de 100*100 BBRs, y se han generado de manera aleatoria diferentes lotes de tareas de 100 tareas cada uno.

Se han seleccionado seis lotes de tareas, con diferentes rangos de tamaño de tarea (el tamaño de las tareas va de 5*5 a 60*60 BBRs), diferentes tiempos de ejecución, llegada, y máximos para fin de ejecución, para poder reproducir diferentes escenarios de cálculo. Como característica general, el lote de tareas 1 tiene predominio de tareas de pequeño tamaño y corta duración, mientras el lote de tareas 6, incluye tareas de tamaño grande y larga duración. El rango de frecuencia de llegada de las tareas se ha elegido para poder garantizar una carga de trabajo sostenible en la FPGA.

Se han utilizado tres parámetros para evaluar los resultados: el volumen total de cálculo rechazado por los algoritmos de gestión, el nivel de ocupación promedio de la FPGA obtenido para cada lote de tareas, y el tiempo correspondiente que necesita el algoritmo para encontrar una solución (para procesar cada lote de tareas).

Los resultados de los dos primeros parámetros se presentan en la figura 5.8, y como era de esperar las heurísticas BF se comportan claramente mejor, especialmente en términos de volumen de cálculo rechazado para lotes de tareas formados por tareas de tamaño mayor y larga duración, como lo son los lotes 5 y 6.

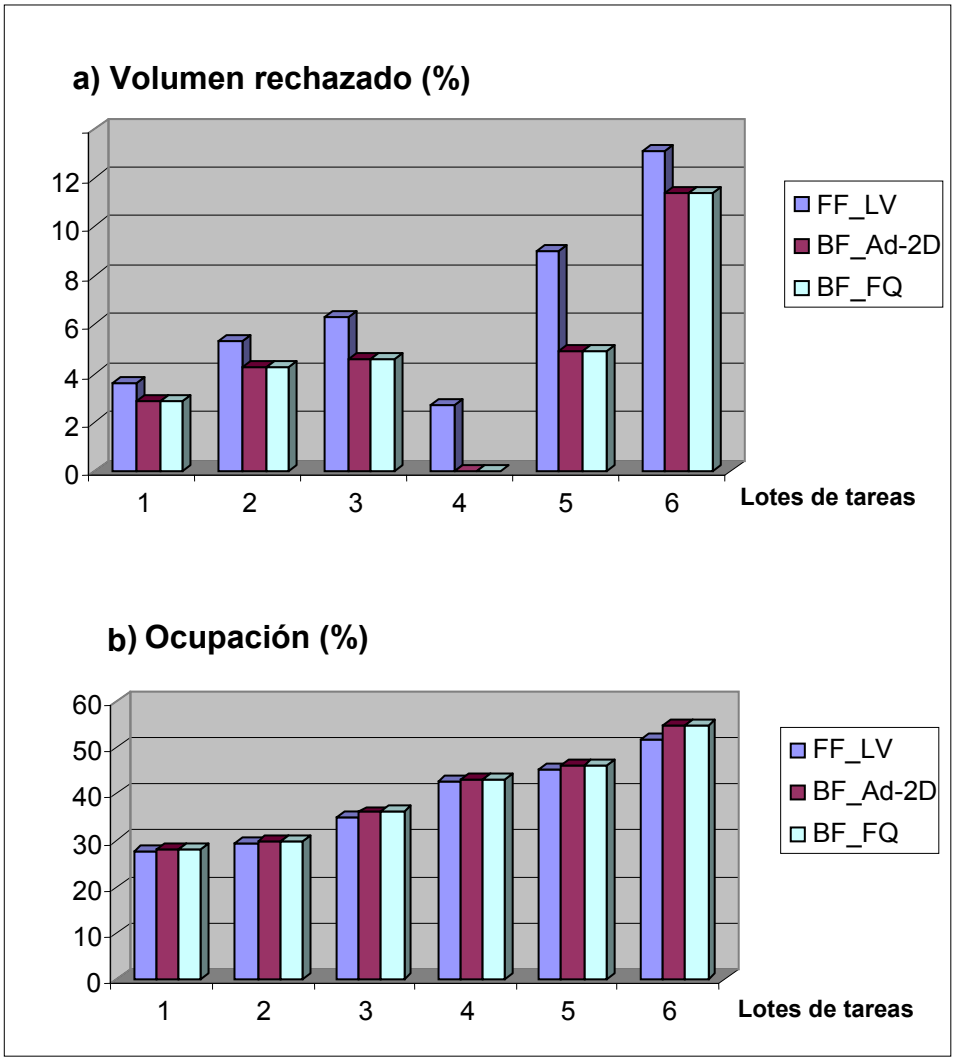


Figura 5.8. Resultados experimentales.

Además se puede comprobar que la heurística BF_FQ proporciona unos resultados muy similares a BF_Ad-2D. Esto es debido a que ambas heurísticas eligen en la mayoría de las situaciones los mismos candidatos para ubicar una tarea dada porque, en general, cuando se consigue el mayor grado de adyacencia entre la tarea y la envolvente del espacio libre, se reduce indirectamente el perímetro del área libre resultante después de insertar la tarea y por tanto la fragmentación estimada con FQ.

En la figura 5.8.b se muestra el nivel de ocupación de la FPGA donde se vuelve a repetir el mismo comportamiento entre las heurísticas BF_FQ y BF_Ad-2D, obteniendo a su vez niveles de ocupación mejores que la heurística simple FF_LV.

Finalmente, ejecutando nuestro simulador en una plataforma Pentium III con SO Windows 2000, a 300Mhz, el tiempo promedio de procesamiento que necesita cada heurística para cada lote de datos, es de un 15% más lento en BF_Ad-2D respecto de la más rápida FF_LV y BF_FQ un 25% más lento con respecto a BF_Ad-2D, como se muestra en la figura 5.9.

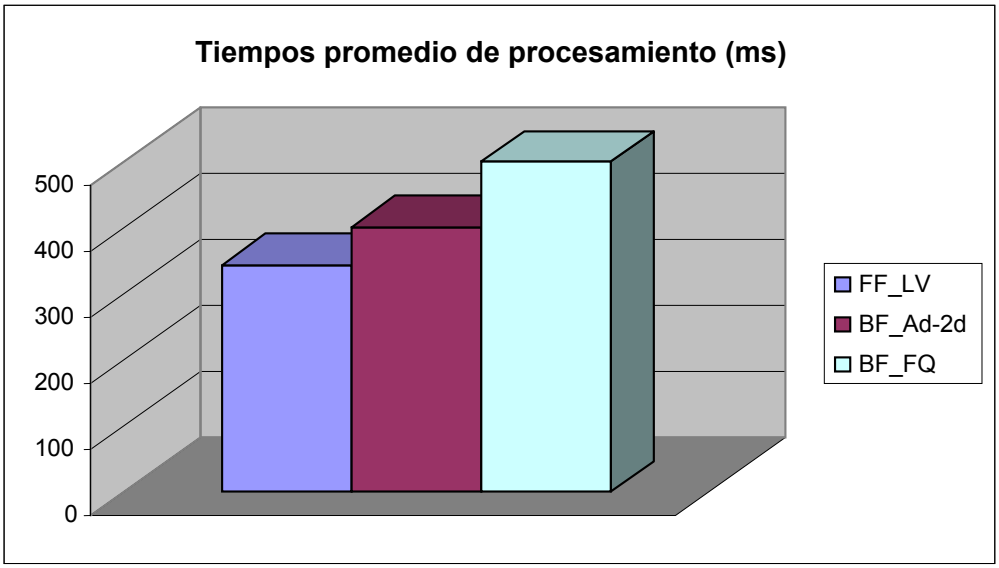


Figura 5.9. Tiempos promedio de procesamiento para cada heurística.

5.4 Conclusiones

Como era de esperar el algoritmo FF_LV es el más rápido porque se queda con el primer candidato donde es viable la asignación de la tarea, pero a costa de un rendimiento peor. En el otro extremo está la heurística BF_FQ con unos tiempos de ejecución sensiblemente mayores. Esto es debido a que esta heurística, cada vez que prueba un candidato, necesita copiar toda la LV y modificarla para ubicar la tarea y poder calcular el nivel de fragmentación. Por contra la heurística BF_Ad-2D no necesita realizar esa copia y es por tanto más rápida.

Por tanto, se puede concluir que la heurística BD_Ad-2D es la más adecuada como heurística de selección 2D, ya que consigue resultados

similares a la BF_FQ pero con tiempos de procesamiento mucho menores, lo que la hace más adecuada para su implementación en un entorno de gestión real.

Teniendo en cuenta que la heurística BF_FQ no permite mejoras relevantes, hay que resaltar que la heurística basada en la adyacencia espacial BF_Ad-2D se puede mejorar y refinar si se consideran parámetros temporales junto con los espaciales, para reducir los niveles de fragmentación en el futuro. Esta solución se desarrollará en la siguiente sección.

Capítulo 6:

Heurísticas de selección de vértices basadas en área y tiempo (3D)

Las heurísticas mostradas en el capítulo 5 realizan sus estimaciones basándose exclusivamente en medidas que se han denominado 2D, obtenidas en un instante determinado. Sin embargo la idea de la adyacencia se podía extender de una forma sencilla considerando el tiempo como tercera coordenada y así tener en cuenta también lo que se prevé que va a ocurrir en un futuro inmediato, durante la ejecución de la tarea. De este modo las tareas se podían representar como cajas 3D en lugar de rectángulos en 2D, cuyo volumen se calcula como en (3.4), y la FPGA como un contenedor de altura infinita.

La figura 6.1 muestra una tarea T_i colocada en la esquina inferior izquierda (posición BL) de una FPGA de 20*20 BBRs. La tarea T_i está caracterizada por la tupla de parámetros definida en (3.1). En 6.1a se muestra la imagen en 2D y en 6.1b en 3D. La FPGA contiene una tarea T_i que configura la envolvente del área libre en 2D (Lista de Vértices) y el área disponible a lo largo del tiempo en 3D. Esta envolvente del espacio libre a lo largo del tiempo está definida por las paredes de la FPGA y las paredes de las tareas adyacentes al espacio libre que están en ejecución (en el ejemplo T_i).

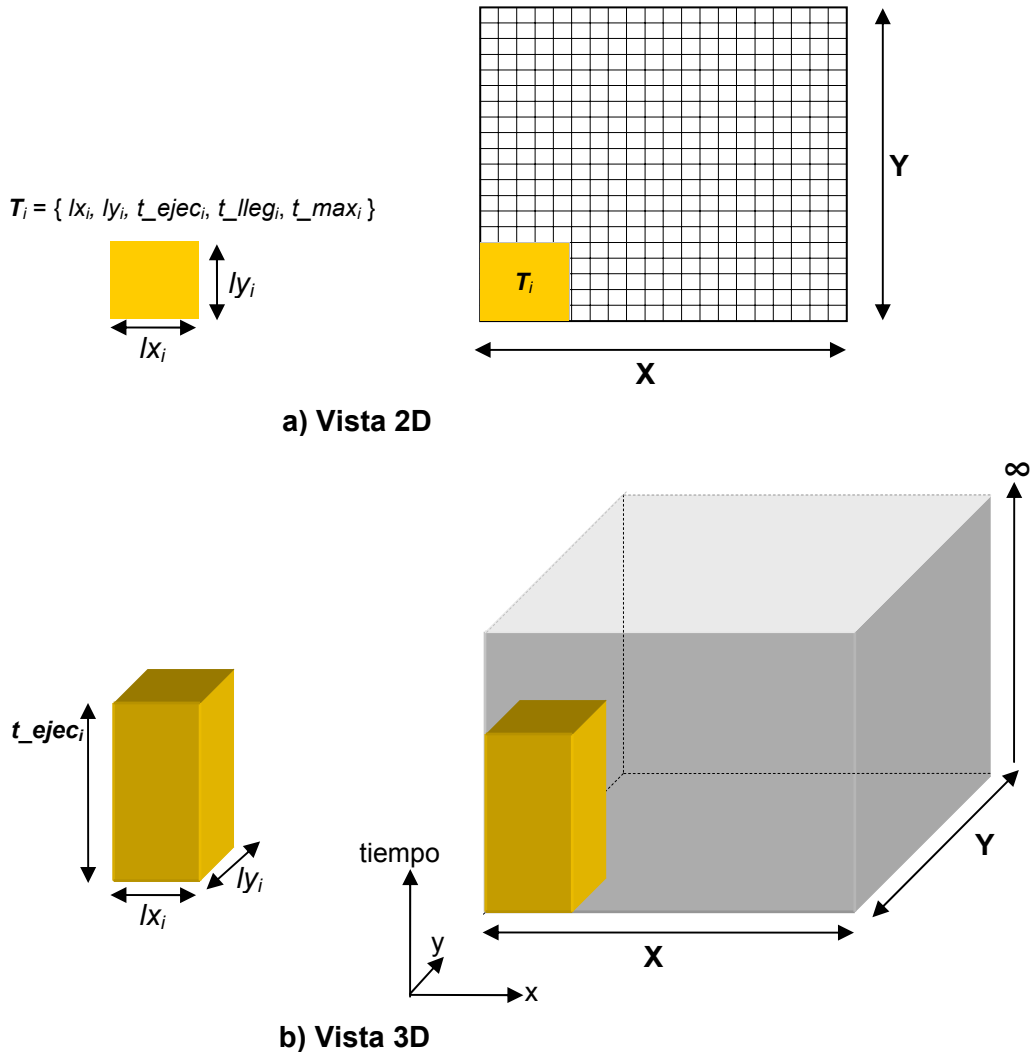


Figura 6.1. Representación de tarea y espacio libre de una FPGA en 2D (a) y 3D (b).

En determinadas situaciones como la mostrada en la figura 6.2 cuando la búsqueda se limita a utilizar la adyacencia 2D como criterio de selección de

ubicación de tarea, el espacio libre queda poco fragmentado en el instante actual, como se puede comprobar en la vista 2D de la FPGA de la figura 6.2.a.

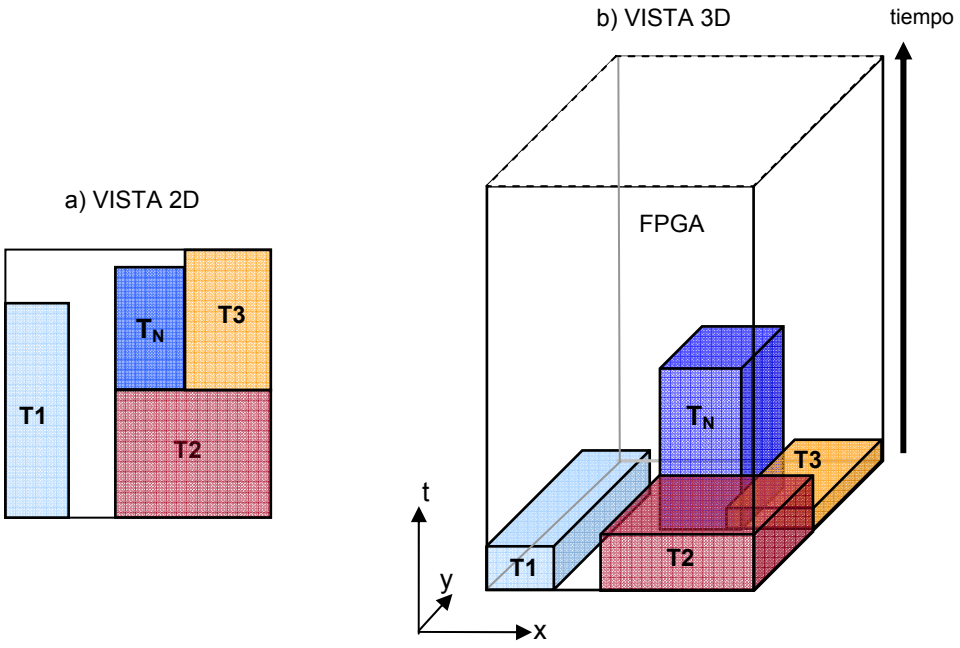


Figura 6.2. Planificación de tarea con heurística de adyacencia 2D con vista 2D (a) y 3D en (b).

Pero a medida que avanza el tiempo y las tareas van finalizando su ejecución, el nivel de fragmentación varía y justo en el instante en que finaliza T_2 (T_1 y T_3 ya han terminado), la nueva tarea T_N queda en el medio de la superficie de la FPGA formando una isla y haciendo que el espacio libre quede muy fragmentado como se observa en la figura 6.3. Si en lugar de utilizar heurísticas basadas en parámetros actuales, se utilizase un criterio que hubiese anticipado los eventos que van a ocurrir en el futuro, esta situación no deseada se hubiese podido evitar.

Sin embargo, la aproximación 3D es difícil de aplicar a las heurísticas basadas en métricas de fragmentación como la BF_FQ, porque la fragmentación debería ser calculada para demasiadas instancias posibles sobre copias del CLV. También se mostrará que la heurística basada en adyacencia 3D lleva a situaciones futuras también con menores niveles de fragmentación.

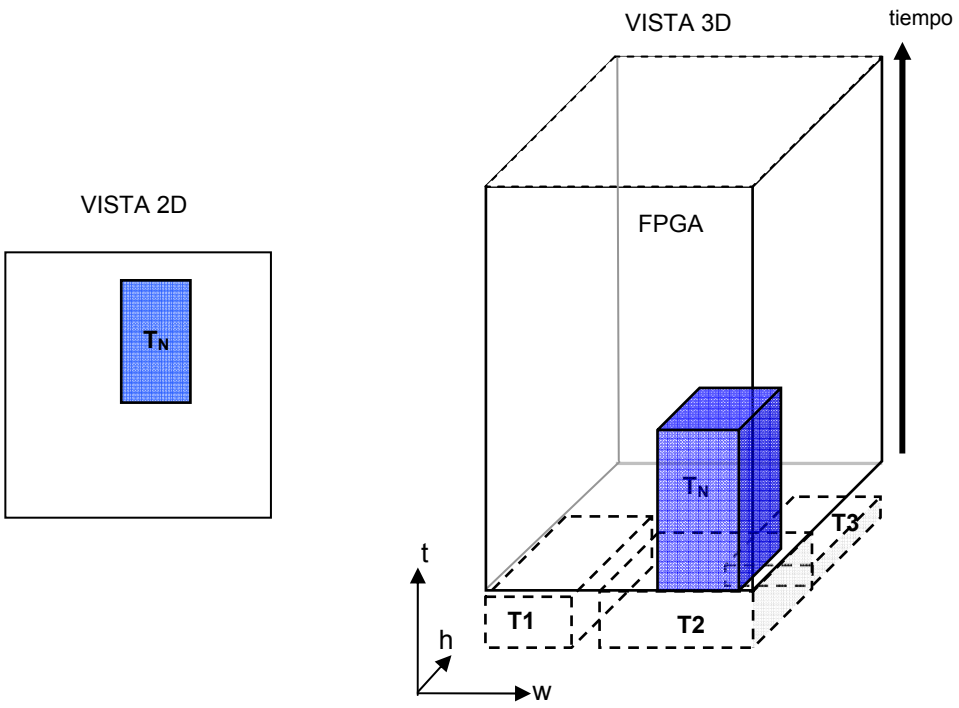


Figura 6.3. Estado futuro de la FPGA con alta fragmentación.

Se puede utilizar como criterio de búsqueda de ubicación de tareas una heurística basada en el concepto de adyacencia espacial 3D y otra más refinada, utilizando el mismo concepto, que amplíe el rango temporal de búsqueda.

Esta aproximación de adyacencia espacial 3D será desarrollada en la siguiente sección y se extenderá a continuación teniendo en cuenta consideraciones de planificación anticipativa (*look-ahead*), dando como resultado una heurística que proporcionará los mejores resultados experimentales.

6.1 Heurística basada en adyacencia espacio-temporal (3D)

Esta heurística es una evolución de la de adyacencia 2D descrita anteriormente, pero extendida al eje del tiempo. Si se considera una FPGA como un contenedor y las tareas como pequeñas cajas con el tiempo en el

eje vertical, este algoritmo basado en adyacencia 3D trata de apilar la caja que corresponde con la nueva tarea $\boldsymbol{T_N}$ lo más cerca posible del resto de las tareas ya ubicadas en el contenedor o de los bordes de dicha FPGA. Esta heurística es similar a las utilizadas en el mundo real para situar cajas dentro de contenedores. La adyacencia 3D por tanto, se estimaría como la superficie total de contacto que hay entre la caja que define la carga de trabajo de la nueva tarea, y la envolvente del espacio libre.

Para tener en cuenta como nuevo parámetro el tiempo, es necesario almacenar un nuevo valor en cada arista de la Lista de Vértices: el valor *t_rest* de la arista de la tarea a la que pertenece, una especie de "tiempo de vida de arista" como se definió en (3.7). Este valor depende del *t_rest* de las tareas que lo forman, excepto para aquellas aristas que son los bordes de la FPGA, donde este valor es ∞ porque son las paredes del contenedor sin limites. Para simplificar y almacenar un único valor reduciendo la búsqueda, si una arista pertenece a más de una tarea contigua, aquella que tenga el menor valor de *t_rest* se elige como tiempo de vida de toda la arista.

El valor de la adyacencia espacio-temporal 3D (Ad-3D) de una nueva tarea $\boldsymbol{T_N}$, calculada para cada vértice candidato V_i, es igual al sumatorio de la longitud del perímetro de la nueva tarea que está en contacto con cada arista *h* de la LV_j de cada hueco que forma el CLV, multiplicado por la adyacencia temporal entre la nueva tarea y la arista *h*. Esta adyacencia temporal es el mínimo entre el "*tiempo de vida"* de la arista $arista\_t\_rest_h$ y el *t_ejec* de la tarea entrante $\boldsymbol{T_N}$:

$$Ad-3D_i = \sum_j \sum_h \left(LV\_arista_{jh} \cap T_N\_frontera\right) * \min\left(arista\_t\_rest_h, t\_ejec_N\right) \quad (6.1)$$

$$\text{con } \mathrm{LV_j} \in \mathrm{CLV} \text{ y } \mathrm{h} \in \mathrm{LV_j}$$

Donde el tiempo de vida de arista, $arista\_t\_rest_h$ se calcula de forma diferente para distintos casos:

a) ∞, si pertenece al borde de FPGA.

b) min ($t\_rest_i \ \forall \ T_i \in$ arista), si pertenece a más de una tarea.

c) t_rest_i , si la arista pertenece sólo a una tarea T_i .

La figura 6.4 muestra los tres casos descritos para determinar el tiempo de vida de arista.

Como se deduce de la fórmula (6.1), cuando la tarea entrante está en contacto con cualquier arista del perímetro de la FPGA, entonces se considera el t_ejec_N de la tarea entrante. De este modo siempre que sea posible, en el tiempo actual, la tarea será preferiblemente situada cerca de los bordes de la FPGA, o bien cerca de otras tareas de larga duración.

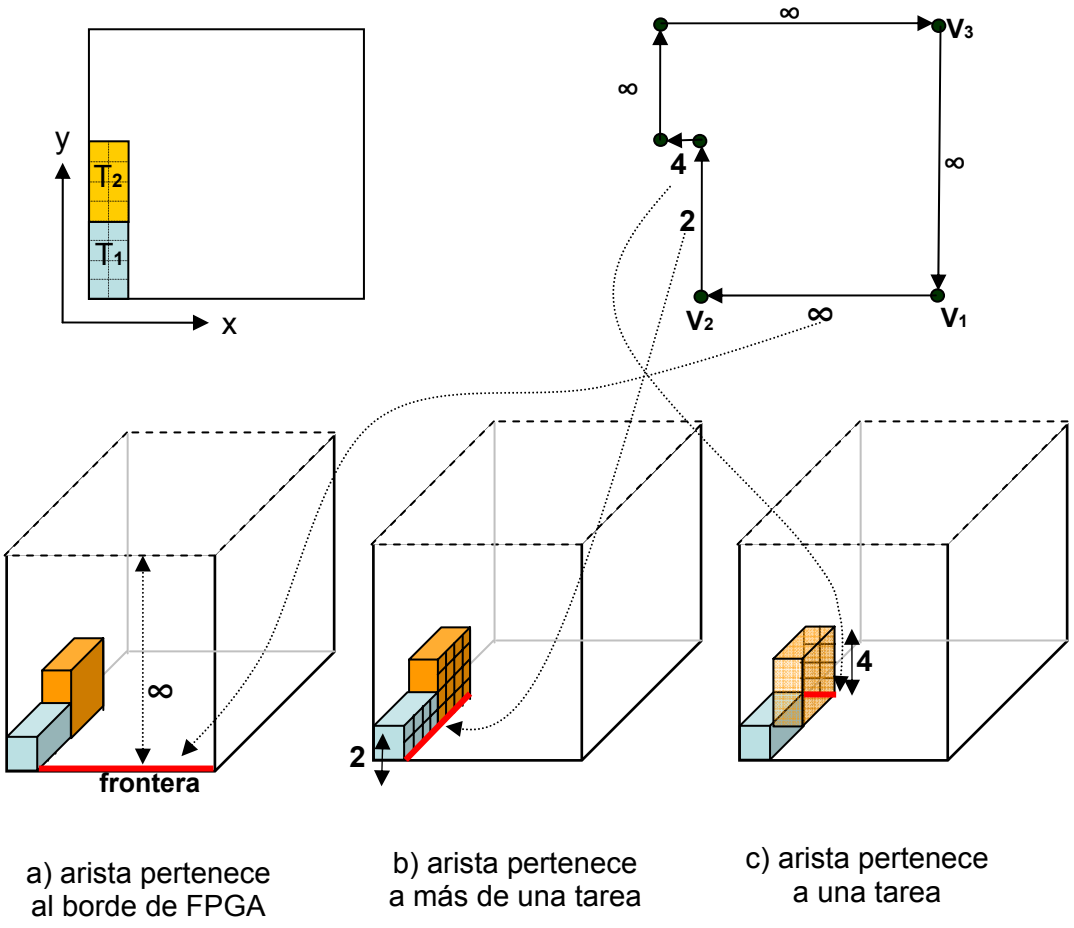


Figura 6.4. Ejemplos de cálculo de tiempo de vida de arista.

La figura 6.5 muestra ejemplos simples que ilustran el método para el cálculo de la adyacencia 3D con una nueva tarea T_N de tamaño 5*5 BBRs y con t_ejec_N = 6, para los tres casos mencionados anteriormente. En la LV aparece cada arista con su tiempo de vida $arista\_t\_rest_h$. Para estos casos la adyacencia 3D se calcula:

Caso a) Adyacencia de T_N en V_1: $Ad\text{-}3D_{N1}$ = (5x6)+(5x6) = 60

Caso b) Adyacencia de T_N en V_2: $Ad\text{-}3D_{N2}$ = (5x6)+(5x2) = 40

Caso c) Adyacencia de T_N en V_3: $Ad\text{-}3D_{N3}$ = (2x4)+(5x6) = 38

En este ejemplo, esta heurística seleccionaría el vértice V_1 para insertar la nueva tarea T_N (el valor de $Ad\text{-}3D_3$ en V_4 es el mismo que en V_1).

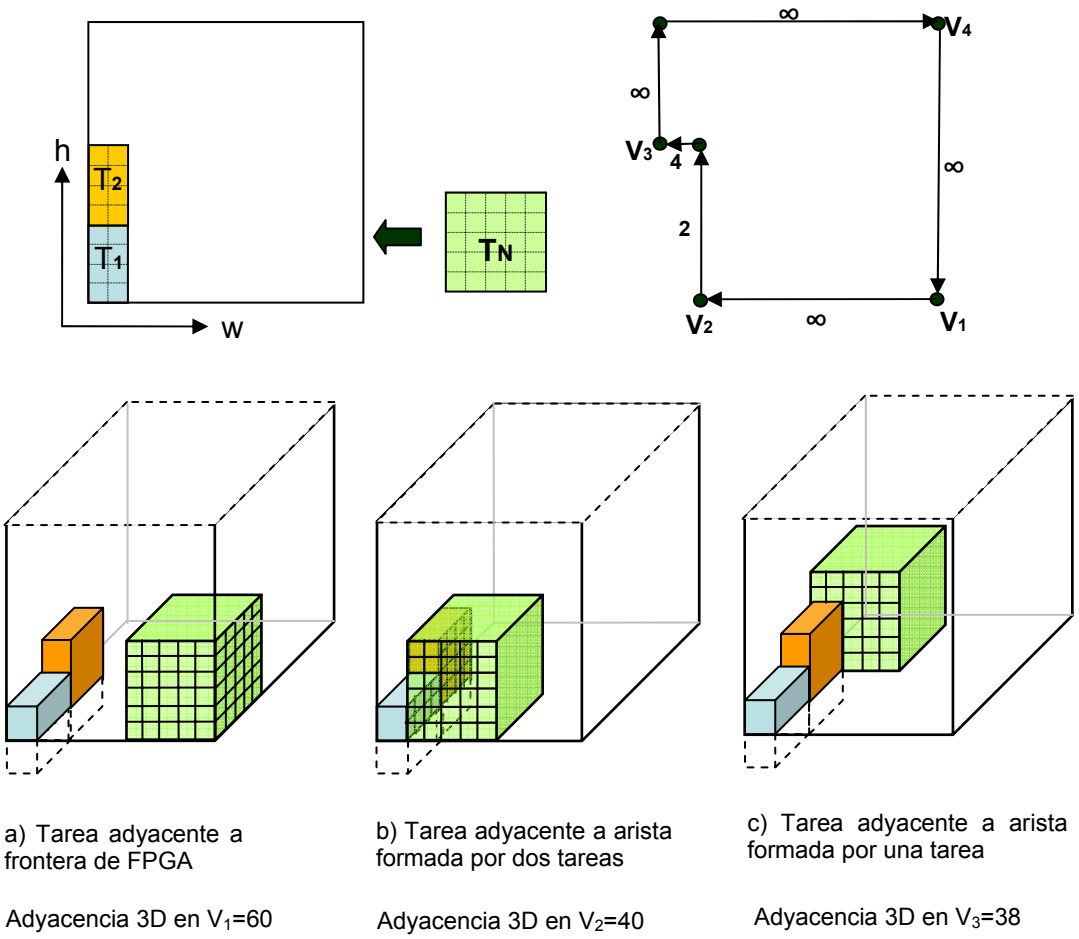
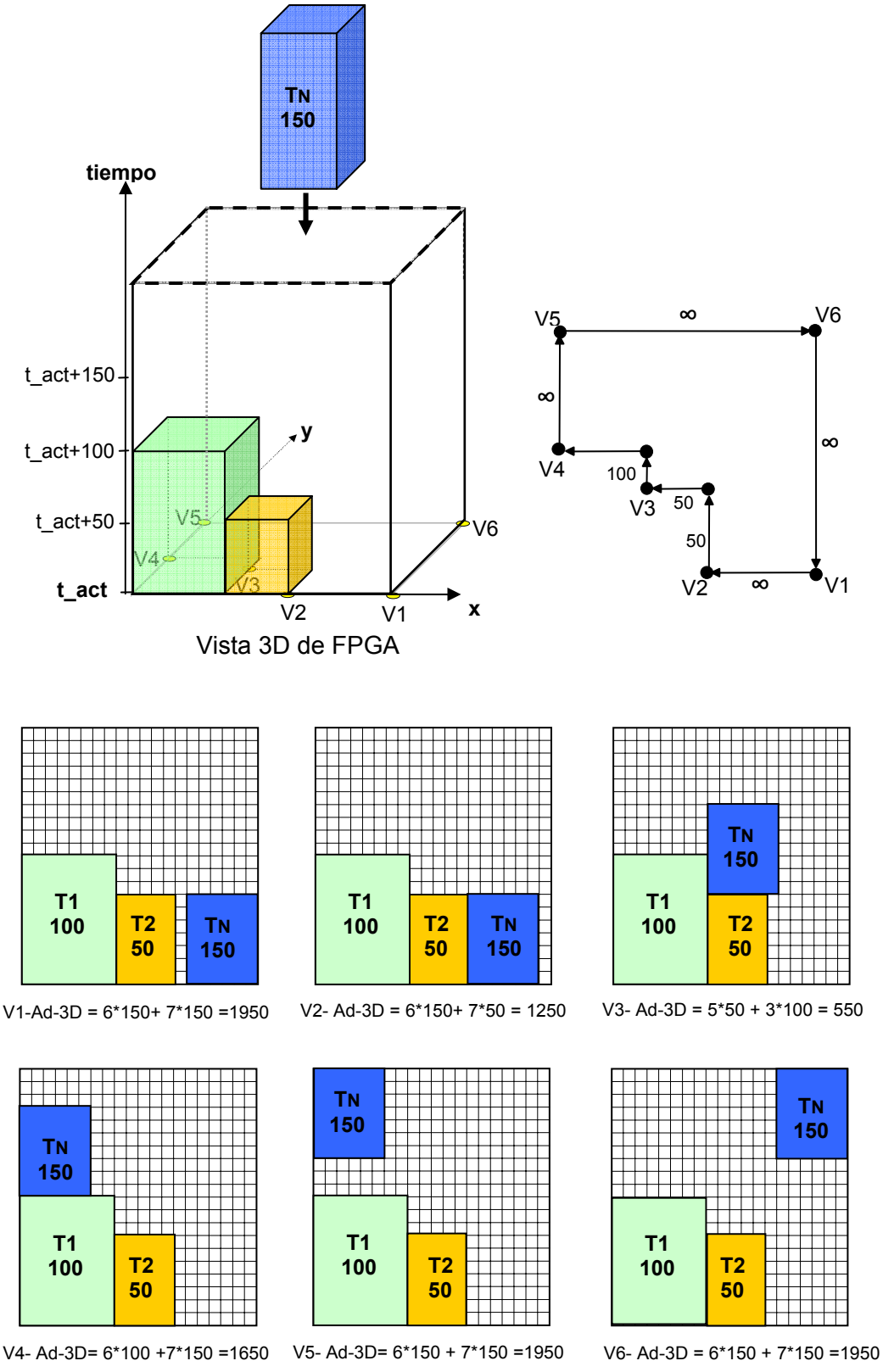


Figura 6.5. Ejemplos de cálculo de adyacencia 3D.

La figura 6.6 muestra una FPGA de 20x20 BBRs con dos tareas en ejecución T_1 (t_rest_1 = 100) y T_2 (t_rest_2 = 50) y una nueva tarea T_N de 6*7 BBRs y t_ejec_N = 150. Además la LV se muestra en la parte superior derecha, donde cada arista aparece con el valor de $arista\_t\_rest_h$ correspondiente. Debajo de cada posible ubicación en la FPGA de la nueva tarea, aparece el cálculo detallado del valor de Ad-3D para todos los vértices candidatos que se realiza como en (6.1).

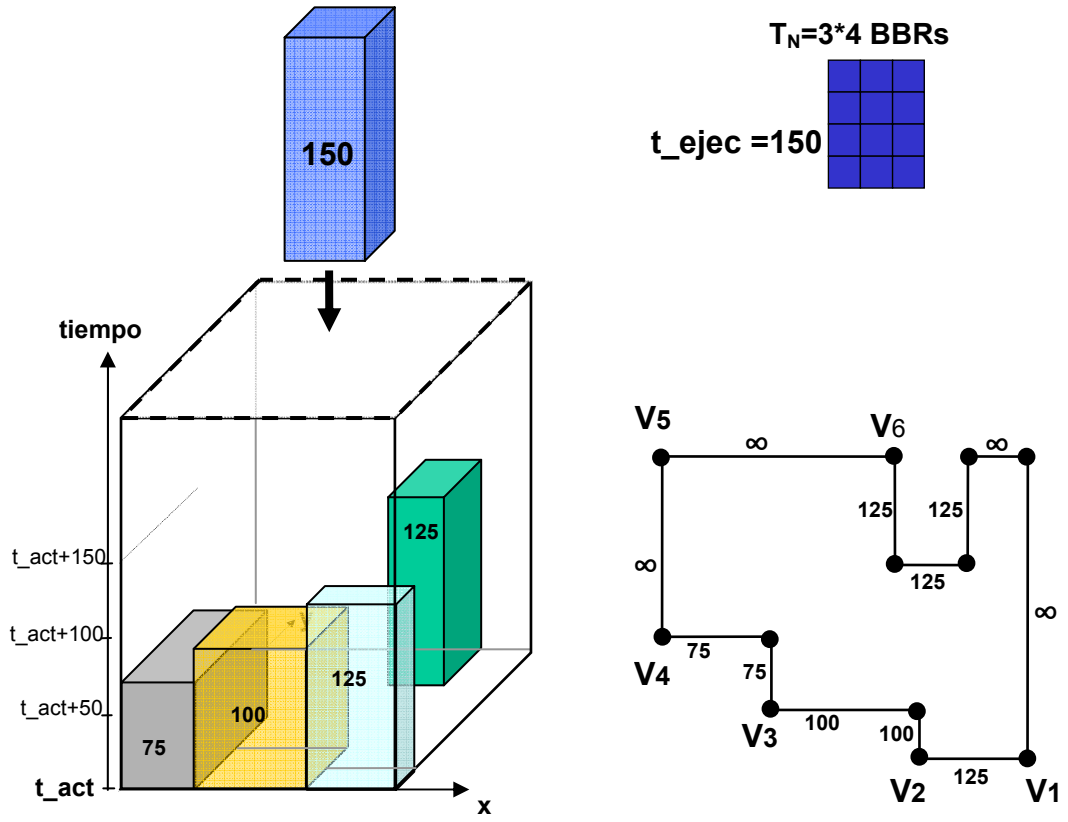


Vista 3D de FPGA

Figura 6.6. Cálculo de heurística Adyacencia 3D para todos los candidatos viables.

Cuando llega la nueva tarea T_N, el Gestor de HW intenta planificarlo en *t_act*. Utilizando la heurística de Adyacencia 3D la nueva tarea T_N se ubicará en el vértice candidato *V1*, el primero donde el valor de Ad-3D es máximo.

La figuras 6.7 y 6.8 muestran un ejemplo simple que ilustra la capacidad de previsión de esta heurística. El estado inicial de la FPGA en 3D se muestra en la figura 6.7.a. Hay cuatro tareas en ejecución, con sus tiempos de ejecución residuales t_rest_i mostrados dentro de las cajas que representan el volumen de cálculo de las tareas. En la figura 6.7.b aparece la LV que define el espacio libre de la FPGA y en cada una de sus aristas aparece el valor correspondiente de $arista\_t\_rest_h$.



a) Estado de FPGA en 3D b) LV

Figura 6.7. Vista de una FPGA en 3D y su Lista de Vértices asociada.

La figura 6.8 muestra en la columna (a) los diferentes estados de la FPGA de la figura 6.7 cuando la tarea T_N de 3*4 BBRs con un valor t_ejec_N de 150 unidades de tiempo se ubica en cada uno de los candidatos viables en el tiempo actual. En las columnas 6.8.b y 6.8.c se muestran la evolución

temporal de los distintos estados cuando van finalizando su ejecución algunas tareas. El valor de la adyacencia 3D para cada candidato viable se muestra debajo de cada estado de la FPGA de la columna (a). Como se puede comprobar esta heurística situaría la nueva tarea en el candidato V_1 (V_2 es equivalente) ya que obtiene un valor máximo de Ad-3D.

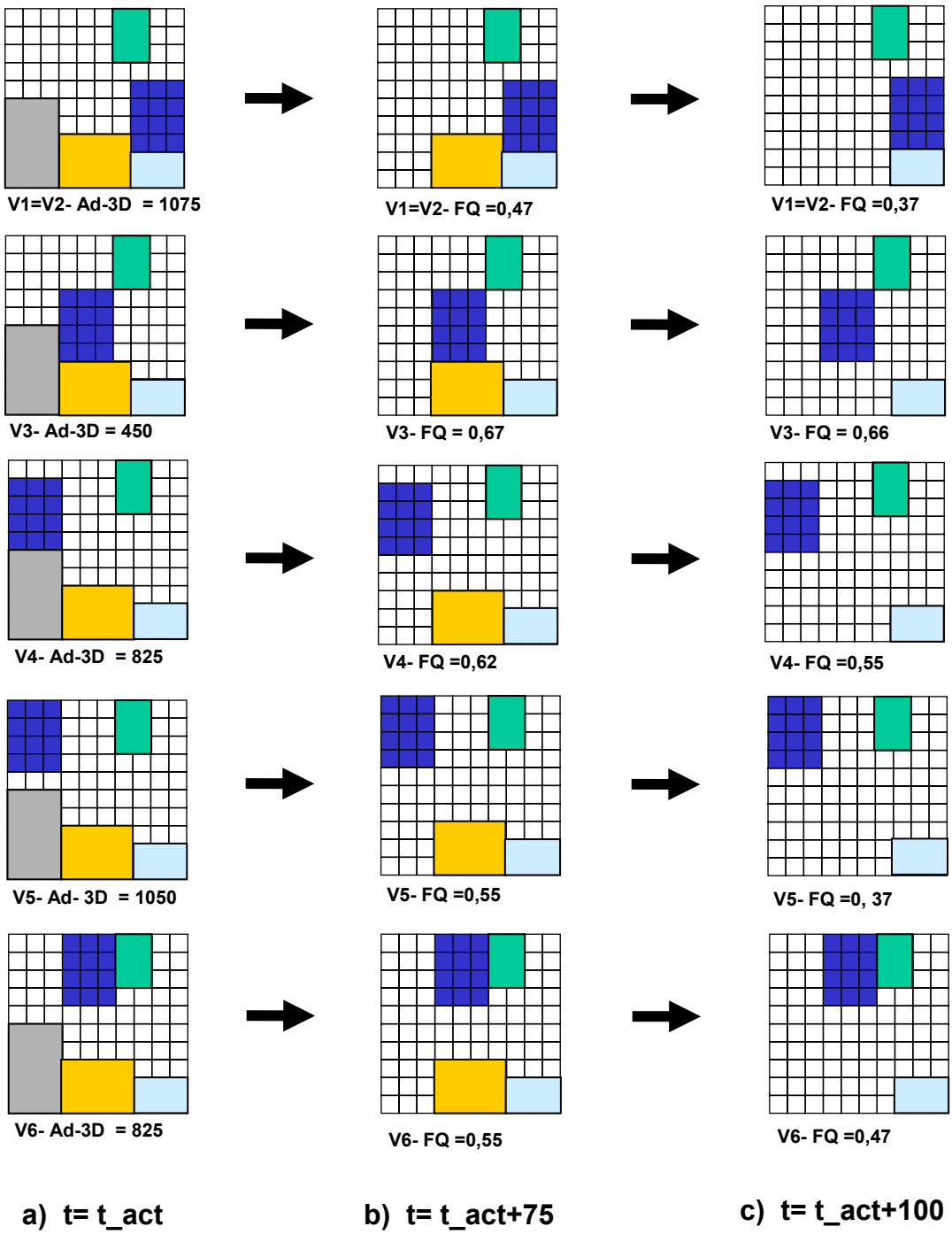


Figura 6.8. Ejemplo de posiciones candidato con heurística Adyacencia 3D.

Además se puede utilizar la métrica de fragmentación FQ para analizar y comparar la calidad de las diferentes alternativas de candidatos. La ubicación de la tarea en el candidato V_1, el que seria seleccionado por la heurística de Adyacencia 3D, lleva a estados futuros de la FPGA con menores valores de fragmentación en los distintos pasos de simulación, como se puede comprobar en la figura 6.8.b y 6.8.c, donde se muestra el valor de FQ en la parte inferior de cada posición candidato viable.

6.2 Heurística 3D anticipativa (Look-Ahead)

Esta heurística es una evolución de la de adyacencia 3D descrita en 6.1, en la que además se tiene en cuenta el próximo evento de finalización de una o varias tareas que sucederá en el futuro.

La figura 6.9 muestra una FPGA en una situación como la mostrada en la figura 6.2 en la que se utiliza una estrategia planificada en lugar de una de decisión inmediata. A diferencia de la heurística anterior, ahora se calcula la adyacencia 3D para ambos, el instante actual y para el próximo evento que corresponde a la finalización de la tarea T_3.

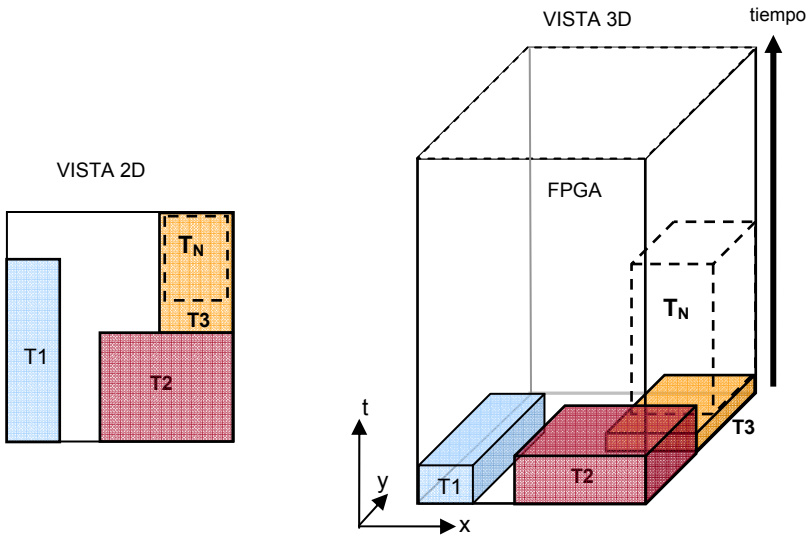


Figura 6.9. Planificación de tarea con heurística de Adyacencia 3D anticipativa.

En este caso, se decide retrasar la ubicación de la nueva tarea T_N hasta que finalice T_3, y de esta forma la fragmentación en el futuro mejora.

En la figura 6.10 se muestran los estados futuros de la FPGA de la figura 6.9, con una representación en 2D (columna izquierda) y en 3D (columna derecha). Estos estados futuros corresponden a eventos de finalización de tareas, donde en (a) finaliza T_3, en (b) finaliza T_1 y en (c) finaliza T_2.

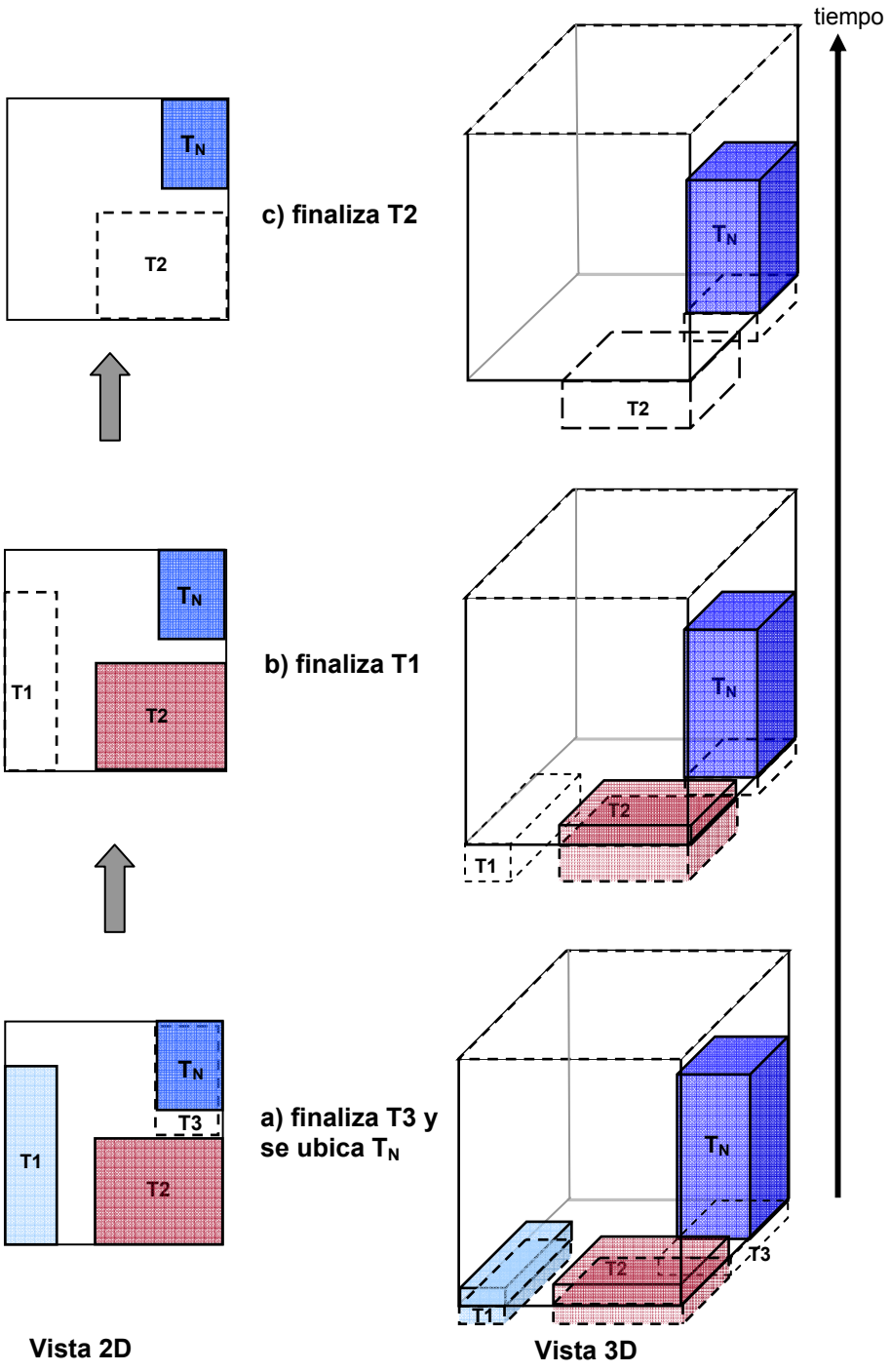


Figura 6.10. Estados futuros de la FPGA con baja fragmentación.

Con esta nueva estrategia cuando finaliza T_2 en (c), la última tarea ubicada T_N no queda en el medio de la superficie de la FPGA formando una isla y el estado de la fragmentación es menor siempre (en a, b y c), a diferencia de lo que ocurría en la situación descrita en las figuras 6.2 y 6.3. En otras situaciones puede no ser necesario retardar la ubicación de la tarea porque los valores de adyacencia 3D en el próximo evento no mejoran.

Resumiendo, esta heurística utiliza el mismo valor de Ad-3D, calculado como se describe en (6.1), pero con esta alternativa dicho valor se calcula para todos los candidatos viables en dos instantes de tiempo diferentes: el tiempo de simulación actual y el tiempo en el que se produzca el próximo evento (cuando finalice la próxima tarea), realizando de esta manera una planificación *Look-Ahead* de un nivel. Sólo se realiza la planificación a un nivel para no introducir demasiada sobrecarga temporal (*overhead)* en las tareas. La idea es que se puede tener un valor de adyacencia 3D mayor para una posición que sólo estará disponible en el futuro, de modo que se amplía el rango de búsqueda en el tiempo.

Sin embargo, algunas veces no es posible realizar esta búsqueda futura si la nueva tarea debe ser planificada antes que cualquier tarea en ejecución pueda abandonar la FPGA, debido al tiempo de arranque límite de la nueva tarea t_lim_i, calculado como en (3.2). Por tanto se debe cumplir la siguiente condición:

$$t\_lim_i > t\_proximo\_evento \qquad (6.2)$$

La figura 6.11 muestra cómo en algunas situaciones la toma de una decisión inmediata es peor que otra decisión planificada, incluso cuando se utiliza la misma heurística para la elección de la ubicación (en este caso ambas estrategias utilizan la heurística de Adyacencia 3D (Ad-3D)). La parte superior de la figura muestra una FPGA con dos tareas en ejecución, representadas en 3D: la FPGA como un contenedor de paredes infinitas (línea punteada) y las tareas representadas por el volumen de cálculo que demandan. Cuando llega una nueva tarea T_N, la toma de decisión planificada decide esperar a que termine la tarea T_2, en t_act+26 y ubicar en la misma posición la nueva tarea. Por contra, la toma de decisión inmediata

representada debajo, tiene que ubicar la nueva tarea en alguna posición que esté disponible en el momento actual *t_act*.

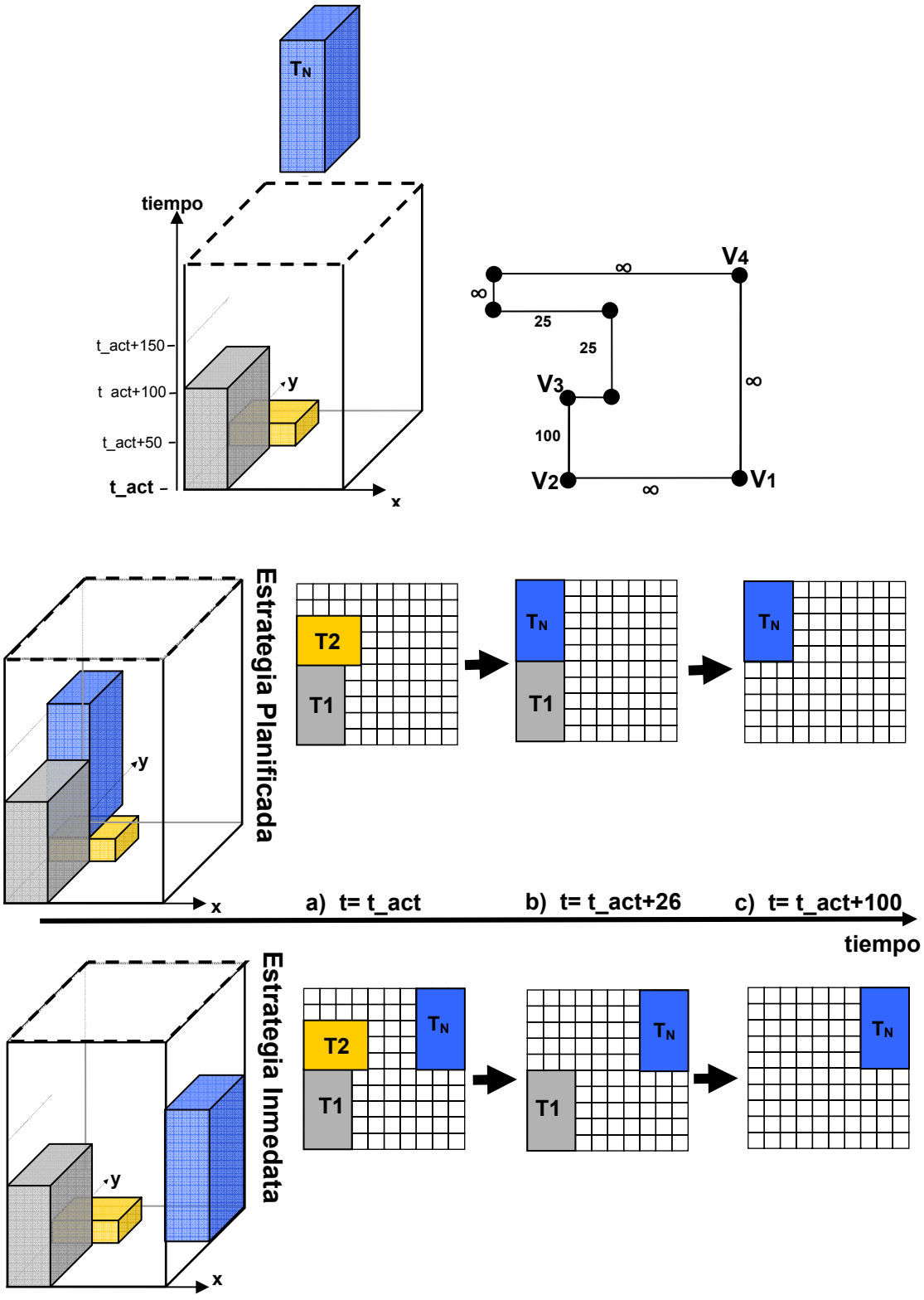


Figura 6.11. Comparación de estrategias de decisión inmediata y retardada.

Para cada una de las decisiones se muestra en horizontal cómo evoluciona en el tiempo y se puede comprobar como la fragmentación es peor con la estrategia inmediata cuando acaba la tarea T_2 en *t_act*+26.

En algunas ocasiones puede suceder que después de haber planificado la ubicación de una tarea en el próximo evento, posteriormente llegue otra nueva tarea muy crítica (debido a que su t_marg_i es muy pequeño) que debe empezar su ejecución muy pronto. En el ejemplo mostrado en la figura 6.10, podría llegar una nueva tarea después de T_N que tenga que ser insertada de manera inmediata debido a su t_marg_i pequeño. En este caso la nueva tarea crítica puede ubicarse sólo si no entra en conflicto con la planificación de T_N.

Como consecuencia de ampliar el espectro de búsqueda, se pueden dar algunas situaciones en las que se puede evitar que se rechacen tareas, como la mostrada en la figura 6.12. Esta figura además muestra una comparación entre el comportamiento de las heurísticas de Adyacencia 3D en (a) y Look-Ahead- Adyacencia 3D en (b). Se ha considerado una FPGA de 20*20 BBRs, con dos tareas en ejecución definidas por la tupla de parámetros T_1={12,12,4,98,110} y T_2={2,13,8,98,118}, situadas en las esquinas BL y TR de la FPGA respectivamente. Con este estado actual de la FPGA, en *t_act* llegan dos nuevas tareas T_3= {6,11,8,100,120} y T_4 ={3,10,15,100,116} de manera simultánea, y se supone que se procesa primero T_3.

Con una estrategia inmediata en (a) que usa la heurística de Adyacencia 3D se obtendría en el instante de simulación actual *t_act*=100, un valor máximo de Ad-3D de 114 para el vértice candidato que está en las coordenadas *X*=18 e *Y*=20. Cuando T_4 se procesa también en *t_act*=100, la heurística Ad-3D no puede encontrar un candidato viable porque T_4 no cabe en ninguna posición de la FPGA y sería enviada a la cola de espera. En el siguiente evento en t=102, aunqueT_4 podría ser ubicada en la posición de coordenadas (0,0), seria rechazada debido a que su valor t_lim_i ya no cumple la condición (3.5).

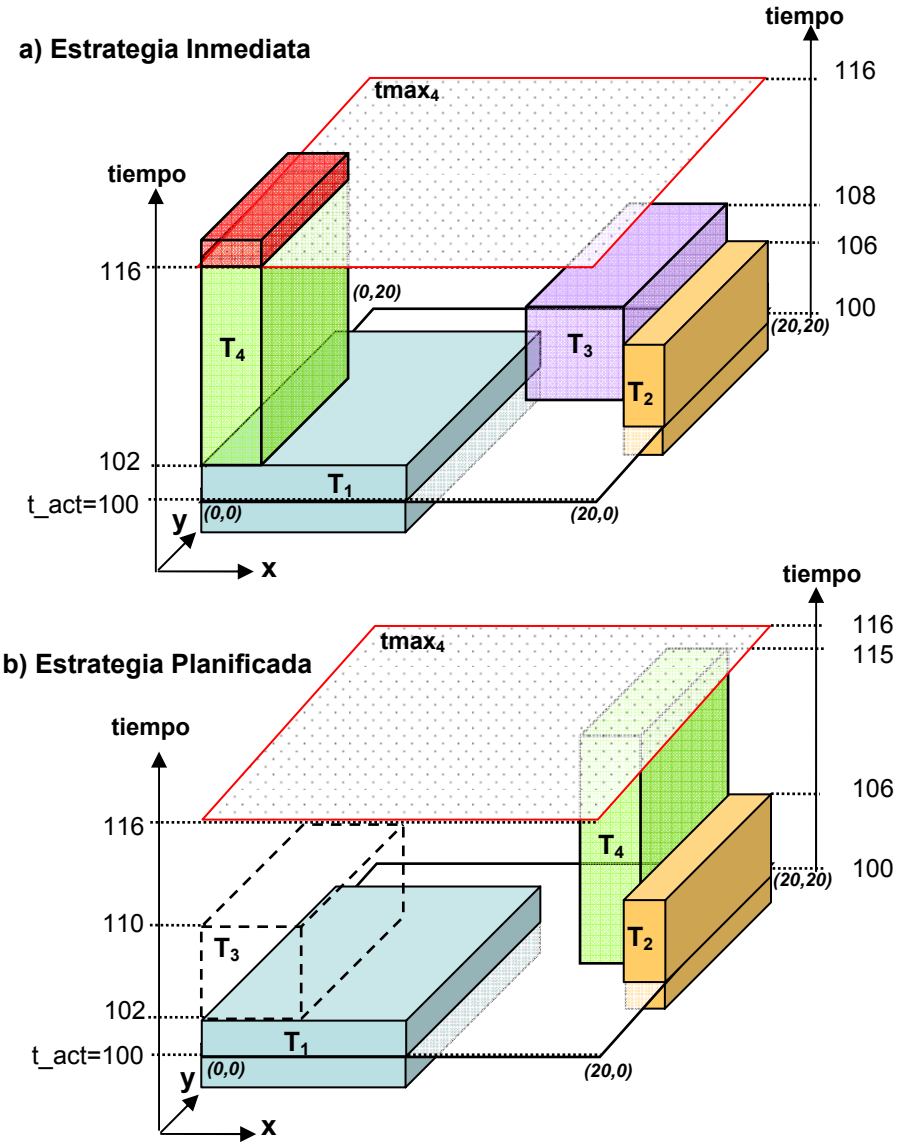


Figura 6.12. Comparación de heurísticas Ad-3D (a) y LA-Ad-3D (b).

Por el contrario, cuando se usa una estrategia planificada mostrada en (b), cuando se procesa T_3 además de obtener el mismo valor de 114 para el vértice candidato en (18,20), calcularía también el valor de Ad-3D para el tiempo en el que se produzca el próximo evento (cuando T_1 finaliza su ejecución). En este caso se obtendría un valor mayor de Ad-3D igual a 136 en las coordenadas (0,0), planificando la ubicación de la tarea T_3 al instante t_inic_3=102. Cuando T_4 se procesa también en t_act=100, ahora sí se encontraría un candidato viable en las coordenadas (18,20) que pueda satisfacer sus restricciones en tiempo real. El plano superior en (a) y (b)

representa el límite temporal impuesto a T_4 por su valor t_max_4 = 116, que en el caso de la estrategia inmediata corta la tarea y no cumple, mientras que con la estrategia planificada se puede cumplir con este requisito.

6.3 Resultados experimentales en 3D

Para evaluar la calidad de las nuevas propuestas, se van a realizar dos grupos de experimentos, uno en el que se tiene en cuenta el parámetro t_max_i de cada tarea (el tiempo máximo permitido para que la tarea finalice su ejecución) en donde se analizará el volumen de cálculo rechazado, y otro en el que no se considera t_max_i, donde se analizará el número total de ciclos usado para ejecutar los diferentes lotes de tareas.

6.3.1 Comparación de heurísticas Best Fit 3D

Se han realizado experimentos usando tres algoritmos diferentes para la gestión del área:

- Lista de Vértices con heurística BF Adyacencia 2D (BF_Ad-2D).
- Lista de Vértices con heurística BF Adyacencia 3D (BF_Ad-3D).
- Lista de Vértices con heurística BF LA- Adyacencia 3D (BF_LA- Ad-3D).

Asimismo, se han generado nuevos lotes de tareas, utilizando los mismos parámetros y criterios que los de la sección 5.2.

En esta comparación se ha incluido también el algoritmo BF_Ad-2D porque tiene el mismo orden de complejidad que BF_Ad-3D. A diferencia de las anteriores, BF_LA-Ad-3D es más lento y necesita casi el doble de tiempo porque tiene que realizar los mismos cálculos dos veces, en el tiempo actual y en el que sucede el próximo evento.

La tabla 6.1 muestra para cada conjunto de datos el número de tareas rechazadas, el nivel de ocupación medio y el volumen de cálculo rechazado por los algoritmos de gestión comparados.

Tabla 6.1. Resultados experimentales.

Conjunto de datos	BF_Ad-2D			BF_Ad-3D			BF_LA-Ad-3D		
	Número de tareas rechazadas	% Ocupación media	Volumen cálculo rechazado(%)	Número de tareas rechazadas	% Ocupación media	Volumen cálculo rechazado(%)	Número de tareas rechazadas	% Ocupación media	Volumen cálculo rechazado(%)
S1	1	24,5	1,5	0	24,9	0	0	24,9	0
S2	4	29,6	5,8	1	30,7	2,4	0	31,45	0
S3	3	32,9	7,8	2	35,2	4,3	0	36,64	0
S4	7	42,14	14,5	2	48	1,9	0	48,7	0
S5	3	60	9,6	1	58	3,7	0	62	0
S6	3	58,28	13,3	1	61	6,1	0	63,5	0

Las figuras 6.13 y 6.14 muestran, de manera detallada, el volumen de cálculo rechazado por los algoritmos y el nivel de ocupación de la FPGA, respectivamente. Los algoritmos BF_Ad-3D y BF_LA-Ad-3D proporcionan mejores rendimientos que el BF_Ad-2D. Además, BF_LA-Ad-3D es capaz de ejecutar todas las tareas, dando un valor de tareas rechazadas de cero para todos los lotes de tareas. También BF_LA-Ad-3D obtiene en la figura 6.14 los mejores niveles de ocupación.

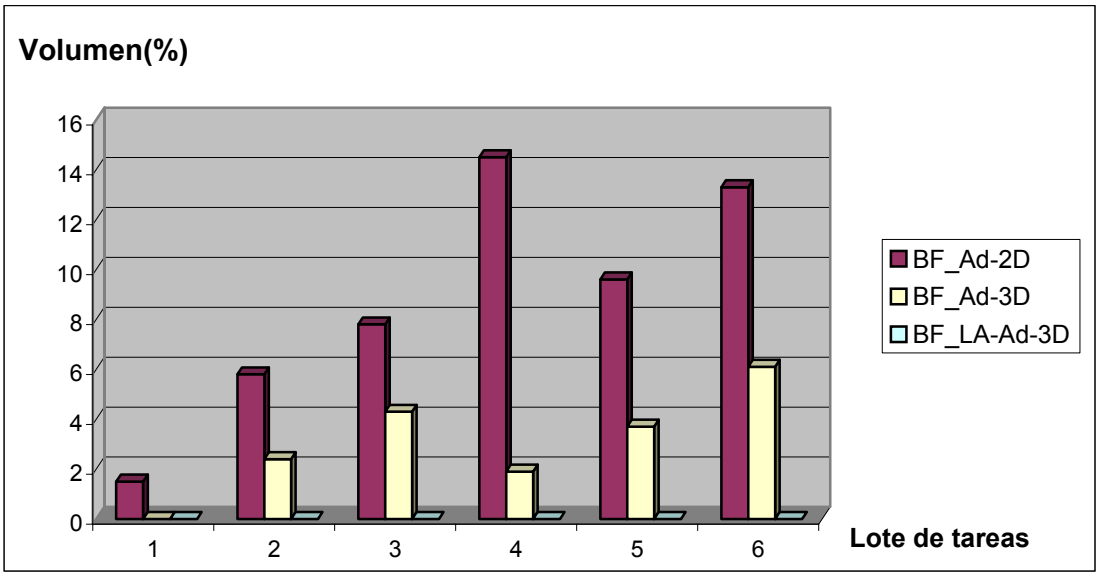


Figura 6.13. Volumen de cálculo rechazado.

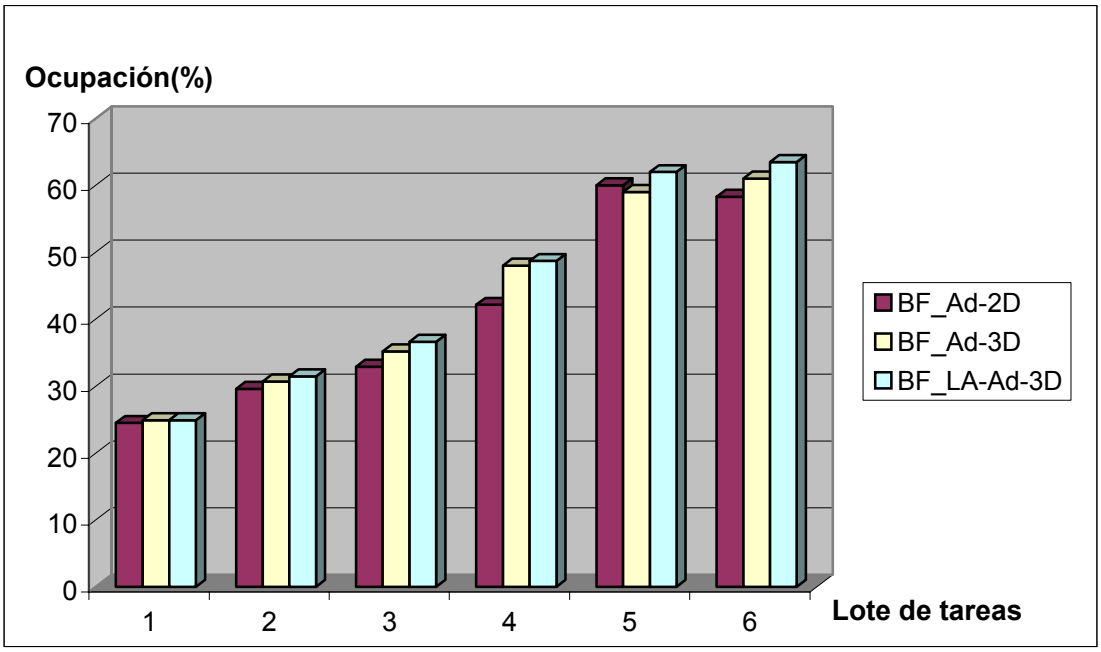


Figura 6.14. Nivel de ocupación en FPGA.

6.3.2 Procesamiento online sin restricciones temporales

Además de los experimentos anteriores, se han realizado otros experimentos con lotes de tareas diferentes, donde el parámetro t_max_i (el tiempo máximo permitido para que la tarea finalice su ejecución) no se considera, y por tanto ninguna tarea es rechazada para su ejecución en la

FPGA. De este modo todos los algoritmos deben realizar la misma cantidad de volumen de cálculo, y ahora el parámetro que se puede utilizar para comparar su eficiencia relativa para gestionar los recursos de la FPGA es el número total de ciclos usado para ejecutar los diferentes lotes de tareas.

Se han generado nuevos lotes de tareas usando los mismos parámetros que en la sección 5.2.2 y los resultados se pueden ver en la tabla 6.2. Se han seleccionado doce lotes de tareas, con diferentes rangos de tamaño de tarea (tamaño de las tareas desde 5*5 hasta 60*60 BBRs)y tiempos de ejecución y llegada para poder reproducir diferentes escenarios de cálculo. Como característica general, los lotes de tareas 1 al 3, tienen predominio de tareas de pequeño tamaño, y una frecuencia baja de llegada de tareas, con lo que se consiguen niveles bajos de ocupación en la FPGA. En el resto de los conjuntos de tareas, se ha aumentado la frecuencia de llegada de tareas, y además incluyen tareas de tamaño grande y larga duración, por lo que se aumenta el nivel de ocupación de la FPGA.

En la tabla 6.2 se muestra el tiempo final, en pasos de simulación, después de procesar todas las tareas. De modo adicional, para los algoritmos BF_Ad-3D y BF_LA-Ad-3D, se indica la mejora máxima en términos de tiempo, comparado con el BF_Ad-2D.

Como se puede comprobar en la tabla 6.2 y en la figura 6.15, para los lotes de tareas 1 al 3, no hay mejora significativa porque los diferentes algoritmos pueden encontrar fácilmente ubicaciones viables para las tareas en espera, debido a dos factores: la frecuencia de llegada baja y el bajo nivel de ocupación en la FPGA. Cuando el nivel de ocupación es alto, los dos algoritmos de Adyacencia 3D mejoran sus rendimientos comparados con el algoritmo BF_Ad-2D. La mejora media para el BF_LA-Ad-3D es ligeramente superior que para el BF_Ad-3D : 9,6% y 9,1% respectivamente.

Tabla 6.2. Resultados experimentales.

Lote de tareas	BF_Ad-2D	BF_Ad-3D		BF_LA-Ad-3D	
	Tiempo total	Tiempo total	Mejora máxima	Tiempo total	Mejora máxima
1	610	610	0%	610	0%
2	605	605	0%	605	0%
3	335	335	0%	333	0,6%
4	347	321	7,4%	321	7,5%
5	366	335	8,5%	335	8,5%
6	296	265	10,5%	247	16,5%
7	239	208	12,9%	196	18%
8	461	405	12,1%	404	12,3%
9	311	272	12,5%	275	11,6%
10	342	291	14,9%	307	10,2%
11	313	255	18,5%	255	18,5%
12	515	454	11,8%	456	11,5%

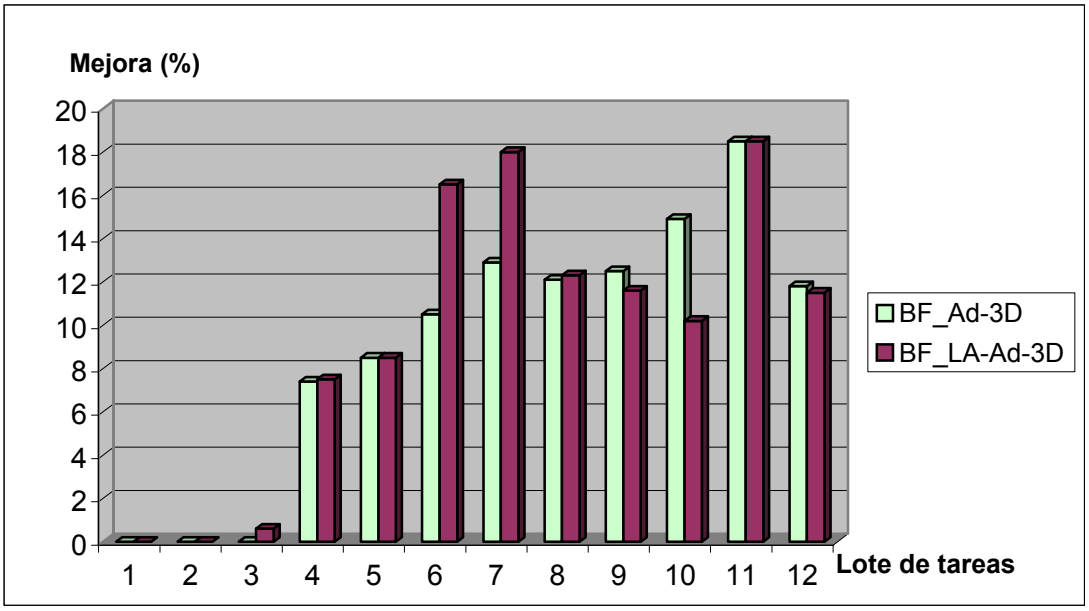


Figura 6.15. Mejora del rendimiento en heurísticas 3D.

6.4 Conclusiones

Con la mejora introducida con la heurística de Adyacencia 3D se obtienen resultados sensiblemente mejores que con anteriores heurísticas basadas en adyacencia espacial, sin incrementar el orden de complejidad. Por otra parte, cuando se utiliza una estrategia más sofisticada, en la que se amplía el rango de búsqueda en el tiempo y se tiene en cuenta el próximo evento de finalización de tarea, se consigue en algunos casos ligeras mejoras, aunque a costa de un mayor coste temporal y por tanto de sobrecarga temporal a las tareas.

La preferencia entre cualquiera de las dos heurísticas basadas en adyacencia espacio temporal BF_Ad-3D y BF_LA-Ad-3D, dependerá del entorno de trabajo y siempre será mas adecuada la primera en aquellos en los que la frecuencia de llegada de tareas sea alta y/o tengan fuertes restricciones temporales (que su t_marg_i sea muy pequeño). Esto es debido a que la heurística BF_LA-Ad-3D es más lenta y supone una mayor sobrecarga en el tiempo de ejecución de las tareas.

Capítulo 7:

Gestión de la Defragmentación

Como ya se ha expuesto en el capítulo 4, la fragmentación del área libre de una FPGA se produce por la ocupación y posterior liberación de los recursos HW disponibles, a medida que van finalizando las tareas. Es por tanto un problema muy frecuente, ineludible y que debe ser controlado y tratado con técnicas que permitan una mejor gestión y aprovechamiento de los recursos de la FPGA.

Aún cuando se utilicen heurísticas inteligentes como las descritas en los capítulos 5 y 6 (incluso la basada en la métrica de fragmentación) para seleccionar la ubicación de cada tarea entrante, es inevitable que aparezcan situaciones donde la fragmentación se convierte en un problema real que tiene que ser tratado. En estas situaciones es inevitable o deseable

suspender la ejecución de tareas y reubicarlas en otras posiciones donde el nivel de fragmentación sea menor.

Para poder defragmentar el área libre disponible en una FPGA con varias tareas en ejecución, se realizarán algunas consideraciones: se supondrá que se dispone de un sistema que tiene los recursos necesarios para **interrumpir** en cualquier momento una tarea que esté actualmente en ejecución, **reubicarla** y **recargar** la configuración de la tarea en una ubicación diferente sin modificar su estado, y que la tarea pueda continuar por tanto con su ejecución desde el mismo estado en que fue interrumpida. Para realizar la gestión de defragmentación se dispone de la misma estructura del Gestor de HW mostrada en la sección 3.3.

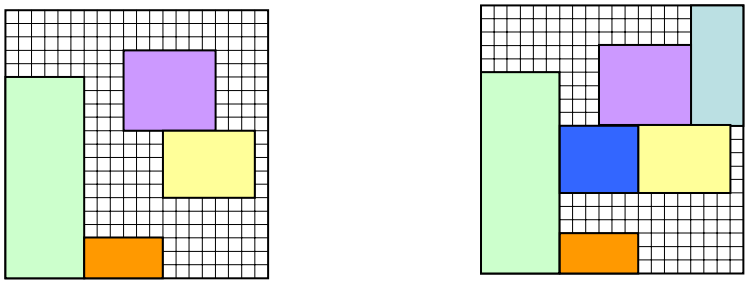
7.1 Estrategias de defragmentación

El *Gestor de defragmentación*, mostrado en la figura 3.1, considerará dos técnicas diferentes de defragmentación, cada una adecuada a diferentes tipos de situaciones:

- ***Defragmentación Preventiva.*** Se iniciará una rutina de *Defragmentación Preventiva* (DP) si el *Analizador de Lista de Vértices* activa una alarma. La *DP* es deseable pero no urgente, y será realizada solamente si las restricciones temporales de las tareas que están actualmente en ejecución no son severas y lo permiten. La DP trata de reducir el nivel de fragmentación para permitir ubicar más tareas en el futuro. Este tipo de alarma tiene dos posibles causas:
  a. La aparición de una isla (formada por una o más tareas) dentro de un hueco de espacio libre, como se muestra en la figura 7.1.a.
  b. La debida a un estado de alta fragmentación en la FPGA detectado por la métrica de fragmentación, como se muestra en la figura 7.1.b.
- ***Defragmentación Urgente.*** Se iniciará una *Defragmentación Urgente* (DU) bajo demanda, si una tarea entrante no puede encontrar una

ubicación viable en la FPGA, aunque haya suficiente espacio libre para poder ubicarla, pero disperso en la FPGA. Dicha defragmentación de emergencia intentará obtener el espacio necesario moviendo una sola tarea de las que se encuentran en ejecución, para retrasar lo menos posible la ejecución de la nueva tarea que espera.

La figura 7.1 muestra dos estados de fragmentación originados por las dos posibles causas descritas en la *Defragmentación Preventiva*, donde se indica el valor de fragmentación calculado con la métrica basada en la cuadratura FQ.



a) Alarma por isla
FQ= 0.77

b) Alarma por alta fragmentación
FQ= 0.76

Figura 7.1. Estados de alto nivel de fragmentación de FPGA.

7.2 Coste-temporal de la defragmentación

Como la defragmentación es un proceso con un alto coste temporal, se necesitará una estimación del tiempo de defragmentación T_D necesario para decidir cuándo, cómo e incluso si se realizará dicha defragmentación.

Hemos supuesto que el coste temporal de defragmentación debido a cada tarea será proporcional al número de BBRs que ocupa la tarea. De ese modo, el coste temporal total de defragmentación será estimado como:

$$T_D = 2 * \sum_i t\_conf_i = 2\boldsymbol{k} * \sum_i (lx_i * ly_i) \qquad (7.1)$$

para todas las tareas T_i en la FPGA que deben ser reubicadas.

El factor de proporcionalidad ***k*** dependerá de la técnica utilizada para reubicar la configuración de la tarea y de las características de la interfaz de configuración (por ejemplo, la interfaz de 8 ó 32-bit SelectMap para FPGAs de las familias Virtex). El factor **2** aparece porque se supone que la recarga de la configuración se realiza para cada tarea a través de la lectura *(readback)* del estado y configuración de tarea desde la ubicación original de la tarea, para copiarlas posteriormente a una nueva posición. Las nuevas ubicaciones para las tareas podrían ser seleccionadas por cualquier heurística de las expuestas en los capítulos 5 y 6.

Para reducir el valor de 2***k***, podemos encontrar propuestas de distinta naturaleza. Dentro de las propuestas teóricas, si la reubicación se pudiese realizar dentro de la FPGA con la ayuda de algunos cambios en la arquitectura, como los propuestos por Compton [CLCK02] en los que se propone la utilización de un *buffer* interno para almacenar momentáneamente las tareas. Sin embargo este *buffer* plantea problemas porque la reubicación de cada tarea tiene que tener en cuenta la ubicación del resto de las tareas en la FPGA y tiene limitaciones ya que sólo es adecuado para FPGAs con arquitecturas 1D.

Otra solución que conseguiría la reducción más significativa del factor 2***k***, sería usando una arquitectura de FPGA con dos contextos diferentes, una versión simplificada de la arquitectura clásica multicontexto propuesta por Trimberger en [TCJW97]. Un segundo contexto permitiría planificar y llevar a cabo una defragmentación global con un mínimo coste temporal. La carga de configuración en el segundo contexto podría realizarse mientras las tareas continúan en ejecución, y tendríamos que añadir solamente el tiempo necesario para transferir el estado de cada tarea actualmente en ejecución desde el contexto activo hacia el otro.

Otra de las propuestas más realistas para la reducción del tiempo de reubicación de tareas es reducir el factor de proporcionalidad 2***k,*** si se evita leer el estado de la tarea o tareas a reubicar. Para poder interrumpir la ejecución de las tareas, se deberían intercalar en determinados puntos de su ejecución unos puntos de chequeo (*checkpoints)* en los que se podría interrumpir salvando el estado, no siendo posible realizarlo en cualquier

momento. De este modo, si las tareas se compilasen en HW obligando a insertar estos puntos de chequeo, no habría que descargar los datos de configuración más los de estado (filtrando posteriormente la información del estado), ya que el estado estaría almacenado en una determinada zona de memoria especifica para cada tarea y sólo dedicada a guardar la información actualizada en los puntos de chequeo. Con esta pequeña modificación se podría reducir prácticamente a la mitad el tiempo de reubicación de una tarea que sería ***k*** más el intervalo de tiempo hasta próximo punto de chequeo.

7.3 Defragmentación Preventiva

Este tipo de proceso de defragmentación se activa a través del módulo *Analizador de LV*, con dos posibles alarmas que la causan: **una alarma de isla** y una **alarma de métrica de fragmentación**. Estas situaciones se corresponden con los ejemplos mostrados en la figura 7.1.a y 7.1.b respectivamente.

La alarma que se comprueba primero es la alarma de isla (formada por una o más tareas que se han quedado aisladas y están totalmente rodeadas de espacio libre). Una isla puede aparecer solamente cuando sucede un evento de finalización de tarea. Resulta obvio que la eliminación de una isla mediante la reubicación de sus tareas lleva a una reducción significativa del valor de fragmentación, y por tanto debe ser tratada de modo específico.

La segunda causa de alarma es aquella en la que el valor de fragmentación de la FPGA supera un determinado nivel de umbral. Esto puede suceder como consecuencia de varios tipos de eventos, y el sistema tratará de realizar, si es posible, una reubicación global o parcial de las tareas que estén en la actualidad en ejecución.

Esta rutina de defragmentación preventiva no es urgente, o al menos no se activa por la necesidad urgente de ubicar una tarea entrante, y su objetivo es reducir de un modo significativo el estado de fragmentación de la FPGA

(siempre que sea posible) realizando alguna de las acciones mencionadas anteriormente.

Este proceso será realizado solamente si el área libre disponible es suficientemente grande, y tratará primero de reubicar islas dentro del hueco libre, o tratará de reubicar la mayoría de las tareas en ejecución si es posible.

7.3.1 Alarma de isla

Aunque la formación de islas no va a aparecer frecuentemente, cuando estas aparezcan dentro de un hueco deben ser tratadas antes de realizar cualquier otra medida para reducir la fragmentación. Una isla dentro de un hueco se representa en el sistema propuesto como una parte de la frontera del hueco, con los vértices de la isla pertenecientes a la LV que define el hueco de igual modo que lo hacen el resto de vértices. Los vértices de la isla se conectan con el resto utilizando dos aristas virtuales, que no representan como lo hacen los vértices normales una frontera real, y por tanto las aristas virtuales no se consideran cuando se comprueban intersecciones de aristas en el proceso de ubicación de tarea.

La figura 7.2.a muestra un ejemplo de una única isla compuesta por dos tareas y su correspondiente LV mostrada en la figura 7.2.b. La alarma de isla es sólo un bit que se activa siempre que el *Analizador de LV* detecta la presencia de un par de aristas virtuales en la LV, que en este ejemplo aparecen como flechas discontinuas.

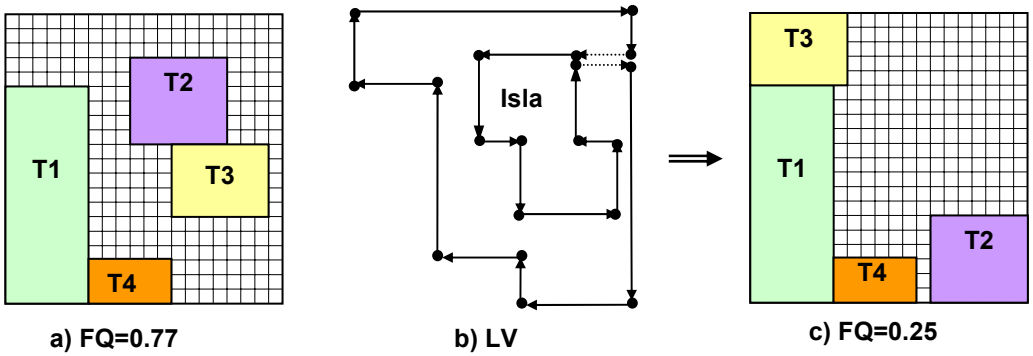


Figura 7.2. Estado de FPGA con una isla (a), su Lista de Vértices (b), y estado después de una defragmentación (c).

Si la alarma de isla se ha activado, primero se chequea si es posible su reubicación o no, demandando que para cada tarea T_i de la isla se cumpla la siguiente condición:

$$t\_marg_i \geq T_{D\_isla} \qquad (7.2)$$

Donde t_marg_i se calcula como en (3.9) y T_{D_isla} es el tiempo necesario para reubicar la isla completa, proporcional al tamaño de la isla y calculado como en (7.1). Si se satisface la condición (7.2), las nuevas ubicaciones de las tareas de la isla son seleccionadas por la heurística BF_Ad-3D explicada en el capítulo 6, aunque podría ser otra heurística la utilizada para tal fin. A continuación hay que actualizar t_marg_i restándole el valor de T_{D_isla}.

Las tareas que se van a mover son siempre reubicadas según sus valores decrecientes de t_rest_i, el tiempo que todavía deben permanecer en ejecución en la FPGA. La figura 7.2 muestra el estado de una FPGA antes en (a) y después en (c) de eliminar una isla formada por dos tareas. Generalmente, la fragmentación después de la eliminación de una isla debe ser sustancialmente menor que el valor de activación de alarma, y de ese modo se puede considerar que se ha llevado a cabo un proceso de defragmentación productivo.

Si la isla no se puede mover porque la condición (7.2) no se cumple, entonces el proceso de defragmentación no se realizará.

7.3.2 Alarma de fragmentación

El *Analizador de LV* comprueba continuamente el estado de fragmentación de la FPGA, estimando su valor con una de las métricas de fragmentación presentadas en el capítulo 4. La alarma de fragmentación se activa siempre que el valor estimado supere un determinado umbral.

Uno de los problemas principales ha sido establecer un valor para este umbral. Para ello, se han considerado las métricas de fragmentación FH y FQ, y se ha analizado su comportamiento en diversas situaciones.

Si se utiliza la métrica de fragmentación basada en la complejidad del hueco FH y calculada como en (4.1) con n=1, se puede observar que para dos huecos rectangulares la fragmentación tiende rápidamente al valor 0,75 o superior. Además, para un hueco simple, la métrica estima valores de fragmentación de 0,6 para una única LV de 10 vértices, o un valor de 0,75 para una LV de 16 vértices. Para los ejemplos mostrados en este trabajo, con un número medio de tareas en ejecución entre cuatro y cinco, se ha seleccionado un valor umbral de **0,75** que correspondería a una LV con un mínimo de 16 vértices, o con varios huecos, resultando el espacio libre altamente fraccionado. Con este valor de umbral, la alarma de fragmentación se activaría en los dos casos representados en la figura 7.1.

Si se utiliza la métrica de fragmentación basada en la cuadratura del perímetro FQ y calculada como en (4.2) para los ejemplos mostrados en este trabajo, se ha seleccionado un valor umbral de fragmentación de **0,6** que correspondería a un hueco *H* (que puede contener islas), cuyo área real sea 0,4 veces el área del cuadrado perfecto A_Q, con el mismo perímetro. De nuevo con este valor de umbral, la alarma de fragmentación se activaría en los dos casos representados en la figura 7.1.

Finalmente, incluso cuando la estimación de la fragmentación alcanza un valor alto, se ha añadido otra condición para decidir si puede empezar el proceso de defragmentación: solamente se puede realizar dicho proceso si el hueco de espacio libre tiene un tamaño significativo.

Se ha establecido un valor de tamaño mínimo de dos veces el tamaño medio de tarea:

$$A_{L\_FPGA} \geq 2 * promedio(A_i) \quad (7.3)$$

Solamente cuando se dan todas las condiciones anteriores, la fragmentación teórica se puede asumir como realmente significativa, y la alarma se activa de manera efectiva.

7.3.3 Defragmentación global inmediata

Si la alarma de alta fragmentación se ha activado, el sistema puede intentar realizar un proceso de defragmentación global en la FPGA. Para poder decidir si es posible tal proceso de defragmentación, se debe comprobar si todas las tareas en ejecución pueden ser reubicadas o no. Para ello es necesario que se cumpla para todas las tareas T_i en ejecución en la FPGA, la siguiente condición:

$$t\_marg_i \geq T_D \quad (7.4)$$

Donde T_D es el tiempo necesario para reubicar todas las tareas en ejecución. El cumplimiento de la condición (7.4) dependerá de las restricciones temporales de las tareas que se encuentran en ejecución en la FPGA y del coste temporal del proceso de defragmentación.

Si todas las tareas satisfacen la condición (7.4), entonces se realiza una **defragmentación global inmediata DGI**, donde todas las tareas en ejecución se reubican empezando desde una FPGA vacía. La configuración y el estado de cada tarea se leen primero, y después se reubican en sus nuevas posiciones como se explicó anteriormente. Para cada tarea se actualiza el valor de t_marg_i restándole el valor de T_D. Para reducir la

probabilidad de que aparezca demasiado pronto una nueva situación de alta fragmentación, las tareas son reubicadas en orden de acuerdo a su valor decreciente de t_rest_i, y la heurística de ubicación de tarea usada es la Ad-3D explicada en el capítulo 6, aunque se podría utilizar cualquier otra heurística. El resultado de este proceso de defragmentación es óptimo.

La figura 7.3 muestra el estado de una FPGA antes y después de realizar el proceso de **DGI**, donde dentro de cada tarea se indica el valor de t_rest_i correspondiente. Se supone en este ejemplo, que todas las tareas cumplen la condición (7.4). La situación inicial de la FPGA es la misma que la mostrada en el ejemplo de alarma por alta fragmentación en la figura 7.1.b y para el conjunto de tareas en ejecución, el coste temporal de la defragmentacion se supone un valor de T_D =20.

Por el contrario, si hay una o más tareas $\boldsymbol{T_j}$ que no cumplen con la condición (7.4), se establece que tienen **restricciones temporales severas** y se denominan **tareas problemáticas**. En este caso, no se puede realizar una DGI y se debe intentar otro tipo de solución. Entonces se establece como una referencia el intervalo definido por el lapso de tiempo entre la llegada de tareas consecutivas, y se denomina tiempo promedio *t_prom*. Dependiendo del instante en el que las tareas problemáticas van a finalizar su ejecución, pueden aparecer dos tipos de situaciones, relacionadas con *t_prom*. Si la condición:

$$t\_rest_j < t\_prom \tag{7.5}$$

se cumple para todas las tareas $\boldsymbol{T_j}$ que no satisfacen (7.4), se puede realizar un proceso de **defragmentación global retardada DGR**, que consiste en reubicar las tareas cuando hayan finalizado las tareas problemáticas. Por el contrario, si (7.5) no se cumple para alguna tarea, se realizará un proceso de **defragmentación parcial inmediata DPI**, que afectará solamente a las **tareas no-problemáticas**. En este proceso todas las tareas en ejecución no-problemáticas son reubicadas empezando desde una FPGA no vacía que contiene las tareas problemáticas en ejecución que no pueden ser reubicadas. El resultado de esta defragmentación no es óptimo, pero sí es inmediato.

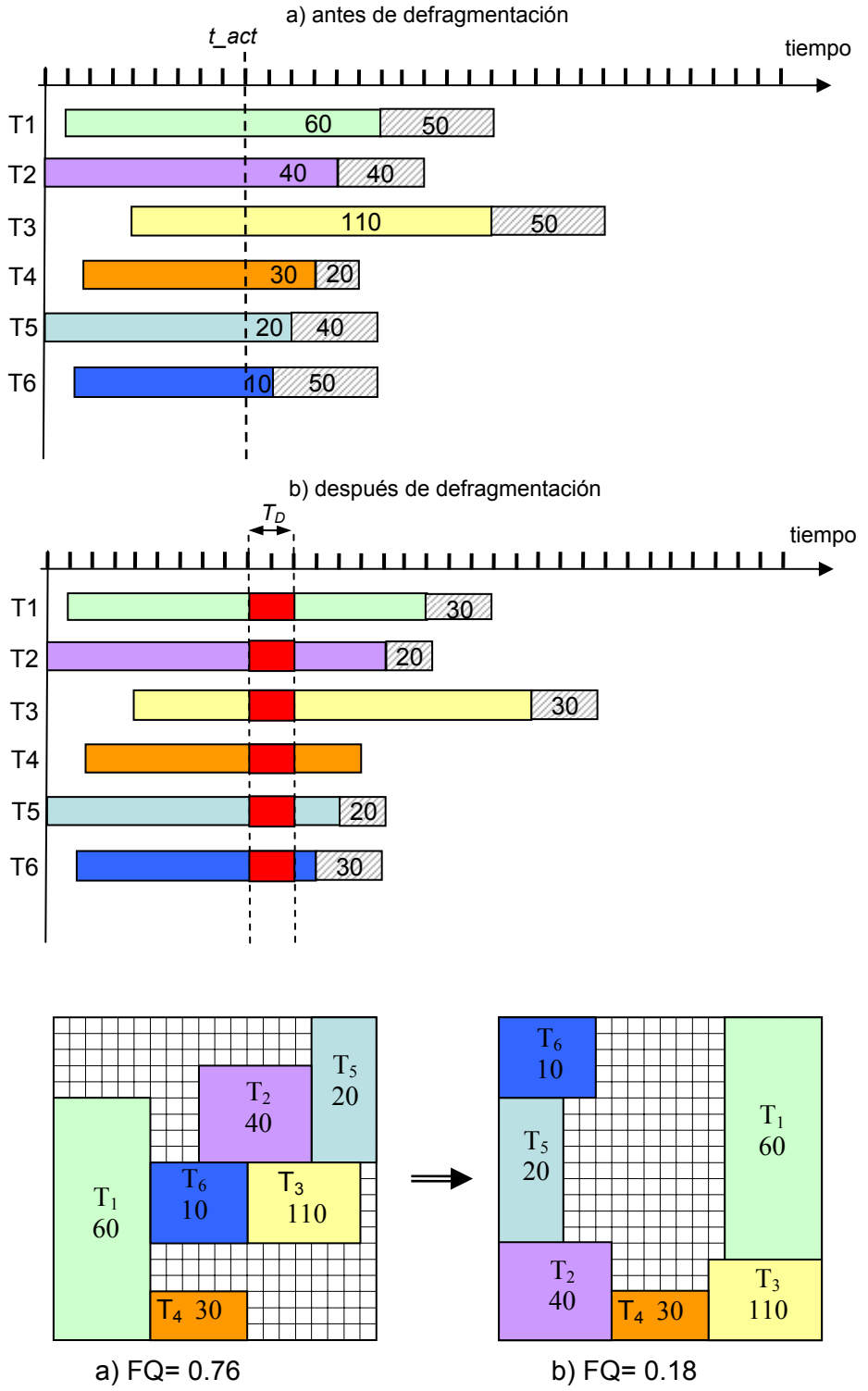


Figura 7.3. Estado de la FPGA antes (a) y después (b), de realizar un proceso de defragmentación global.

7.3.4 Defragmentación global retardada

La heurística de defragmentación global retardada **DGR** se utiliza cuando la condición (7.5) se cumple para todas las tareas T_j que no satisfacen (7.4), es decir, cuando la tarea o tareas T_j con restricciones temporales severas van a acabar "pronto". Si todas las tareas problemáticas finalizan antes de que se alcance este umbral de referencia, entonces se puede esperar un tiempo hasta que acabe la última y realizar una defragmentación global para el resto de tareas, que no son problemáticas. En este caso, el resultado de la fragmentación es óptimo, pero no es inmediato, y pueden llegan nuevas tareas que deben ser tratadas antes del inicio del proceso de defragmentación. Mientras dura este proceso de defragmentación no se puede atender a demandas de las nuevas tareas que pueden ir llegando. Las tareas que puedan llegar durante este lapso de tiempo (probablemente un número reducido) serán directamente copiadas a C_{esp}, en el caso de que no tengan restricciones temporales severas. En caso contrario, si llega una tarea que tenga restricciones temporales severas, el proceso de defragmentación se suspende inmediatamente para poder gestionar la nueva tarea.

La figura 7.4 muestra una situación derivada también de la figura 7.1.b, donde la condición (7.4) no se cumple ahora para la tarea T_6, aunque dicha tarea problemática si cumple la condición (7.5) porque termina pronto su ejecución y por tanto no se puede realizar un proceso de defragmentación global inmediatamente en *t_act*. La situación descrita en 7.4.b corresponde al instante de tiempo después de 10 ciclos cuando T_6 ha finalizado su ejecución. Además se supone que no han llegado nuevas tareas después de T_6. La figura muestra cómo es posible obtener un valor de fragmentación mejor en 7.4.c, aunque no de manera inmediata.

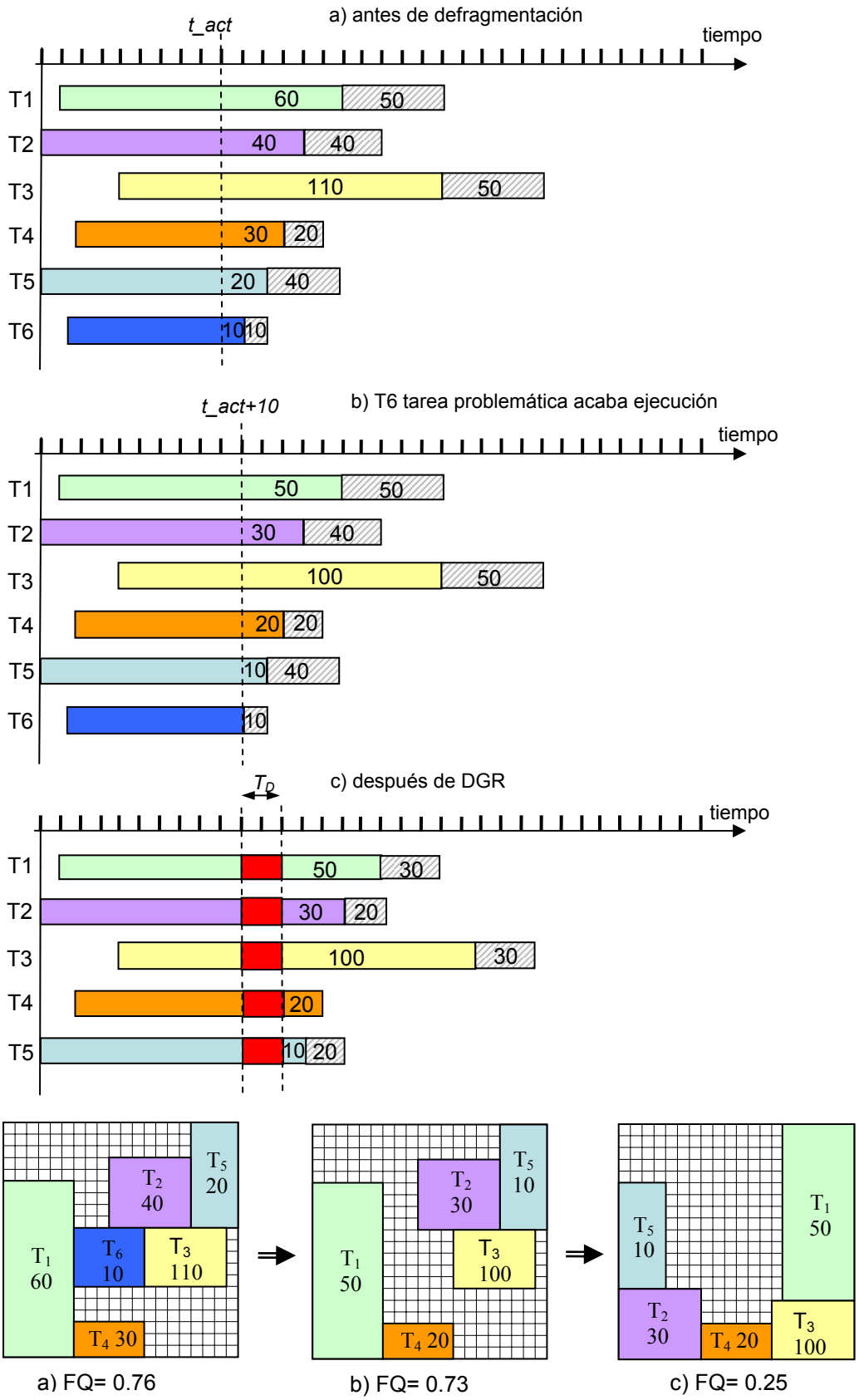


Figura 7.4. Estado de FPGA antes (a), espera (b) y después (c) de realizar defragmentación global retardada.

7.3.5. Defragmentación parcial inmediata

La heurística de defragmentación parcial inmediata **DPI** se elige si las tareas problemáticas, con restricciones temporales severas van a terminar su ejecución "tarde", es decir, la condición (7.5) no se cumple. En este caso se realiza de manera inmediata un proceso de defragmentación parcial, mediante la reubicación de todas las tareas, excepto aquellas que son problemáticas. Este tipo de defragmentación no es óptimo, pero puede reducir muy pronto de manera significativa el valor de fragmentación. Las configuraciones de tareas que van a ser reubicadas son leídas (*readback)* y reubicadas como en un proceso de Defragmentación Global, pero en lugar de empezar con una FPGA vacía, esta debe incluir de partida todas las tareas problemáticas.

La figura 7.5.a muestra la misma situación inicial de la FPGA que la figura 7.1b, pero ahora la condición (7.4) no se cumple para la tarea T_6 (se supone que la tarea no acaba pronto su ejecución), y por tanto no puede ser movida interrumpiendo su ejecución. La figura 7.5 es similar a la 7.4, pero cambia en la tarea T_6 el valor t_rem_6= *60 en lugar de 10,* lo que condiciona el tipo de estrategia de defragmentación a emplear. El estado de fragmentación resultante en 7.5.b no es tan bueno como el obtenido con la fragmentación global retardada de la figura 7.4, pero es inmediato.

Como comentario general se debe resaltar que en todos los ejemplos mostrados en las figuras 7.3, 7.4, y 7.5 el nivel de fragmentación se reduce de manera notable.

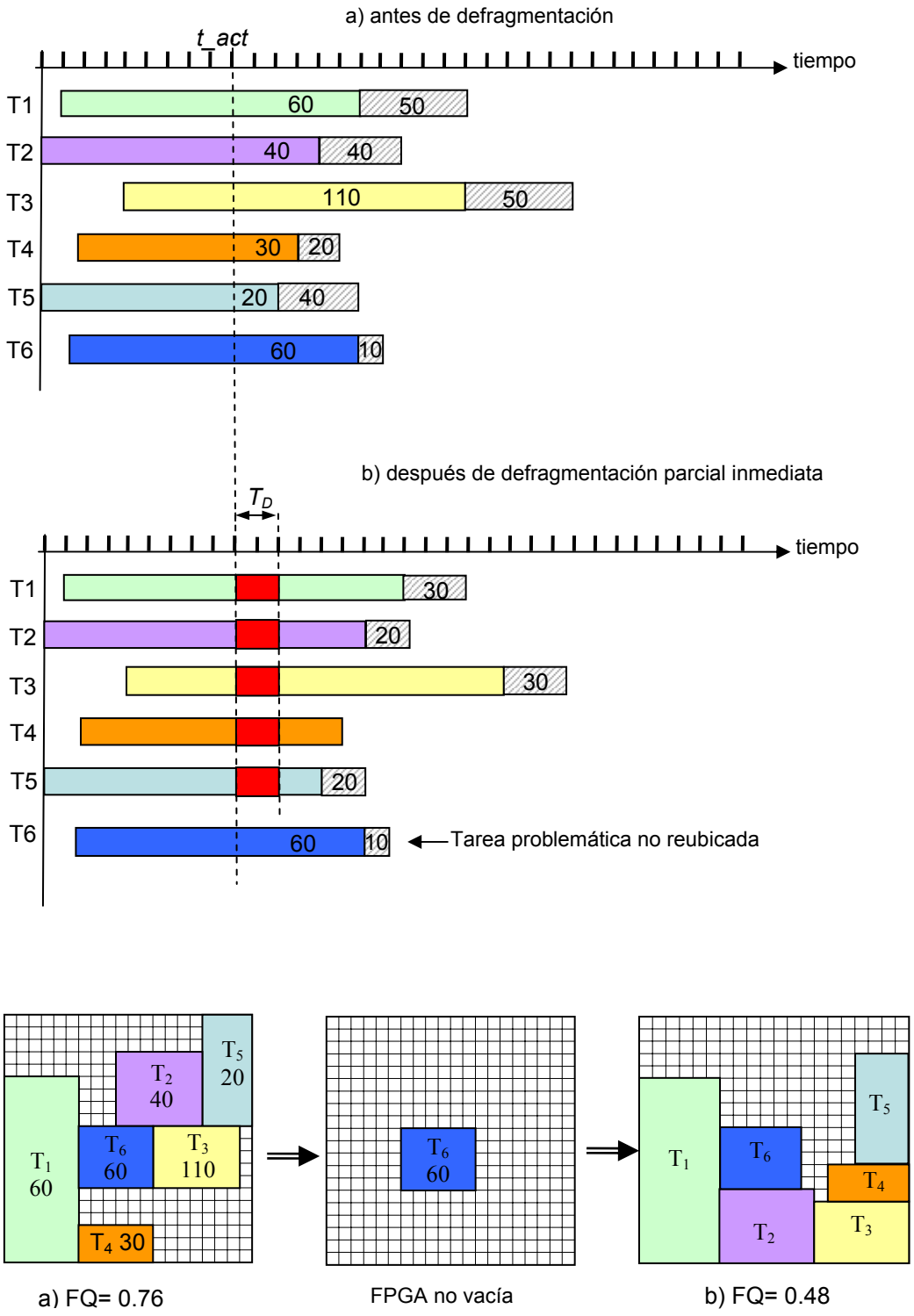


Figura 7.5. Estado de FPGA antes (a) y después (b), de realizar defragmentación parcial inmediata.

7.4. Defragmentación Urgente

Este tipo de defragmentación sólo se llevará a cabo en una situación de urgencia, cuando una nueva tarea T_N no puede ser ubicada dentro de la FPGA debido a la fragmentación del espacio libre, a pesar de todas las medidas preventivas anteriormente explicadas. Entre los posibles motivos que puedan generar este tipo de fallo se encuentra la presencia de muchas tareas problemáticas (con restricciones temporales severas) en la FPGA, o un nivel de fragmentación alto pero por debajo del umbral de alarma.

En este caso se intenta mover una única tarea para obtener suficiente espacio libre contiguo como para alojar la nueva tarea, pero solamente si el problema es la fragmentación y no la falta de espacio libre. Esto se comprueba tomando acciones para defragmentar sólo si el área libre disponible en la FPGA es dos veces el área de la tarea entrante, y que debe cumplir la condición:

$$A_{L\_FPGA} \geq 2 * (lx_N * ly_N) \tag{7.6}$$

siendo lx_N y ly_N las dimensiones de la tarea.

Si se cumple la condición (7.6), se elije como **mejor tarea candidato para reubicación** T_R, la tarea T_i con el mayor porcentaje de su perímetro $P_i=2(lx_i+ly_i)$ perteneciente al perímetro total, que denominamos su **adyacencia relativa** $Ad\text{-}R_i$, y que además realmente se pueda mover.

El módulo *Gestor de defragmentación* calcula el valor $Ad\text{-}R_i$, para cada tarea en todo el perímetro como:

$$Ad\text{-}R_i = \left[\frac{P_i \cap LV}{P_i} \right] \tag{7.7}$$

El algoritmo de ubicación de tarea se mantiene al tanto de tal candidato a reubicar, cada vez que se modifica la LV, considerando solamente valores de $Ad\text{-}R_i$ mayores de 0,5. De este modo, cualquier tarea cuyo perímetro coincida en su mayoría con el perímetro del área libre, daría un alto valor de $Ad\text{-}R_i$, próximo a 1. Conviene matizar que una tarea en forma de isla no puede ser elegida como tarea candidato a reubicación porque habría disparado otro tipo

de alarma especifica (ver alarma de isla en 7.3.1). El caso límite serían las tareas “unidas” al resto del perímetro solamente por uno de sus vértices, o aquellas que podríamos denominar “cuasi-islas” como la tarea T_1 mostrada en la figura 7.6. Es necesario aclarar que la cuasi-isla es diferente a la isla, ya que no está unida al perímetro por aristas virtuales y además no está totalmente rodeada de espacio libre.

La figura 7.6 muestra un ejemplo de cómo se calcula la adyacencia relativa para dos tareas, donde una de ellas T_1 es un caso especial de cuasi-isla y por consiguiente tiene el valor máximo de $Ad\text{-}R_i$.

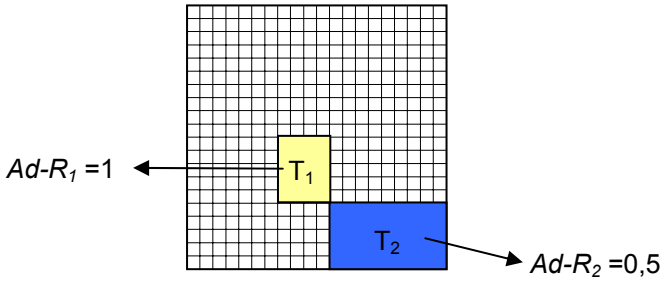


Figura 7.6. Ejemplo de cálculo de adyacencia relativa.

Más aún, $\boldsymbol{T_R}$ debe satisfacer:

$$t\_marg_R \geq T_{DR} \tag{7.8}$$

siendo T_{DR} el tiempo de reubicación de la tarea candidato $\boldsymbol{T_R}$.

Además, la tarea entrante $\boldsymbol{T_N}$ también debe tener un margen temporal suficiente para poder esperar a que se reubique la tarea $\boldsymbol{T_R}$. Por tanto debe satisfacer una condición similar:

$$t\_marg_N \geq T_{DR} \tag{7.9}$$

Si se cumplen estas dos condiciones, $\boldsymbol{T_R}$ es reubicada con la heurística de adyacencia-3D (3D-Adj), y entonces se vuelve a considerar la tarea entrante $\boldsymbol{T_N}$, para buscar de nuevo una posición candidato viable con la nueva disposición de tareas.

La figura 7.7 muestra un ejemplo de cómo se desarrolla una defragmentación urgente, donde aparece una FPGA con un alto grado de fragmentación y llega una nueva tarea $\boldsymbol{T_N}$. En esta situación se dan todas las condiciones explicadas anteriormente para realizar una defragmentación urgente. La figura 7.7.a muestra el candidato $\boldsymbol{T_R}$, con un valor $Ad\text{-}R_i$ de 0,92. En 7.7.b aparece la FPGA después de reubicar la tarea $\boldsymbol{T_R}$, y en la figura 7.7.c aparece la FPGA después de asignar una posición a la nueva tarea $\boldsymbol{T_N}$. En la situación que aparece en 7.7.c, donde todas las tareas tienen un valor de $Ad\text{-}R_i$ de 0,5 o menor, no hay un candidato $\boldsymbol{T_R}$ disponible ya que no hay un movimiento de tarea único que sea ventajoso y efectivo para reducir la fragmentación.

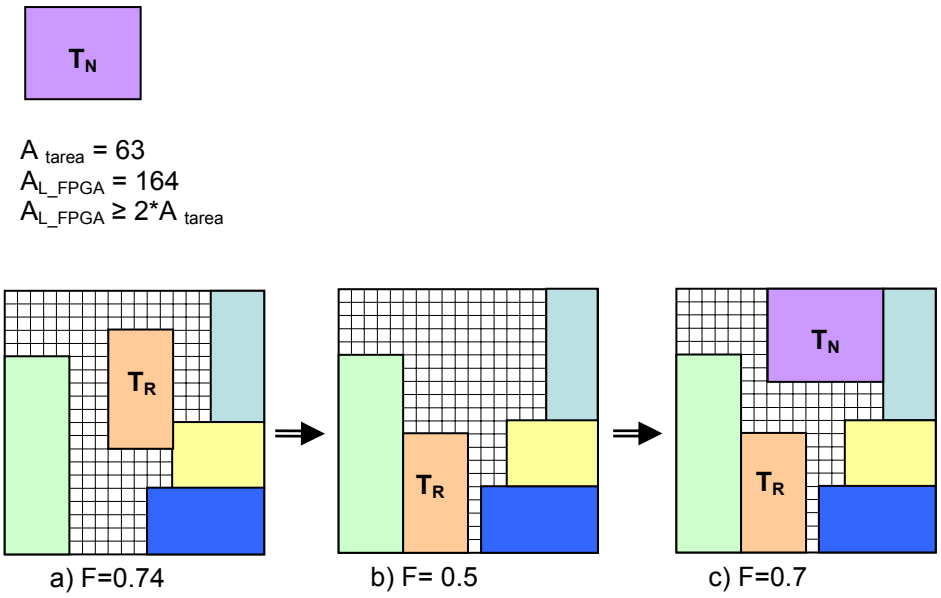


Figura 7.7. Estado de FPGA antes (a), y después (b, seguido de c) de una defragmentación urgente.

Si no hay un candidato válido $\boldsymbol{T_R}$, entonces no se realizará la fragmentación bajo demanda y la tarea $\boldsymbol{T_N}$ será enviada directamente a C_{esp}, en espera de una oportunidad en el futuro antes de que se agote su t_marg_N. Sucede lo mismo si la defragmentación no proporciona los resultados esperados.

La figura 7.8 muestra el pseudocódigo del Gestor de HW mostrado en el capítulo 2, donde se ha añadido la parte de gestión de Defragmentación.

```
While (no_llega_nueva_tarea T_N)
{
      While TareaAcabaEjecucion( t_act)
      {
            ActualizarL_ejec(PlanificadordeTarea)
            ActualizarHuecos(ActualizadorDeListaDeVertice)
      }
      If HayTareasEnEspera(C_esp)
      For (pTarea=C_esp.begin();pTarea!= C_esp.end(); pTarea++)
      {
            If (TareaEspera_Timeout)
                  RechazarTareaParaEjecucionHW
            Else
            {
            If UbicaciónTareaEsperaViable(SelectorDeVertice)
                  {
                  InsertarTareaEspera(L_ejec)
                  EliminarTareaEspera(C_esp)
                  CargarTareaEsperaenFPGA(Cargador/ExtractorTarea)
                  ActualizarHuecos(ActualizadorDeListaDeVertice)
                  }
            }
      }
      t_act+1
}
If UbicaciónViableTareaNuevaT_N(SelectorDeVertice)
      {
      InsertarTareaNuevaT_N(L_ejec)
      CargarTareaNuevaenFPGA(Cargador/ExtractorTarea)
      ActualizarHuecos(ActualizadorDeListaDeVertice)
      }
Else
      If (fragmentaciónBajoDemandaPosible)&(T_N_es_critica)
            {
            IdentificarTareaT_R
            If (T_R cumple t_marg_R ≥ t_DR,)
                  {
                  ReubicarTR
                  InsertarT_N
                  }
            }
      Else PonerEnEsperaTareaNuevaT_N(C_espCesp)
```

Figura 7.8. Pseudocódigo del Gestor de HW que incluye parte de la gestión de defragmentacion urgente bajo demanda.

La parte correspondiente a la gestión preventiva de la defragmentación será ejecutada en el módulo Gestor de defragmentación del Gestor de HW. El pseudocódigo se muestra en la figura 7.9. Las condición denominada *(ISLA_se_puede_mover)* es cierta si se cumple la condición (7.2), la

condición *(NoHayTareasProblematicas)* se cumple si (7.4) y *(TareasProblematicasAcabanPronto)* si se cumple la condición (7.5).

```
Gestor de Defragmentacion

If  (LVi tieneISLA)&(ISLA_se_puede_mover)
      MoverIsla //utiliza puntero para reducir complejidad N a 1.
Else If  (EstadoAltaFragmentación)
       {
            If (NoHayTareasProblematicas)
                  DefragmentacionGlobalInmediata
            Else If (TareasProblematicasAcabanPronto)
                        DefragmentacionQuasiglobal
                  Else DefragmentacionGlobalRetardada
      }
```

Figura 7.9. Pseudocódigo del Gestor de HW que incluye parte de la gestión de defragmentacion preventiva.

7.5 Resultados experimentales

Para evaluar la calidad de las técnicas de defragmentación propuestas, se ha realizado un experimento con una FPGA de 100*100 BBRs, y se han generado nuevos lotes de tareas, utilizando los mismos parámetros y criterios que los de la sección 5.2.

Se han seleccionado cinco lotes de tareas donde se generan las distintas situaciones expuestas en la sección 7.1, para poder aplicar las técnicas de Defragmentacion Preventiva (activada por la aparición de una isla o un estado de alta fragmentación) y de Defragmentación Urgente.

Para realizar el experimento se han usado dos algoritmos diferentes:

- Lista de Vértices con heurística BF Adyacencia 3D (BF_Ad-3D).
- Lista de Vértices con heurística BF Adyacencia 3D y técnicas de defragmentación (BF_Ad-3D_Defrag).

La figura 7.10 muestra el volumen de cálculo rechazado, y como era de esperar la heuristica BF Adyacencia 3D y técnicas de defragmentación se

comporta claramente mejor, logrando ejecutar todas las tareas de todos los lotes.

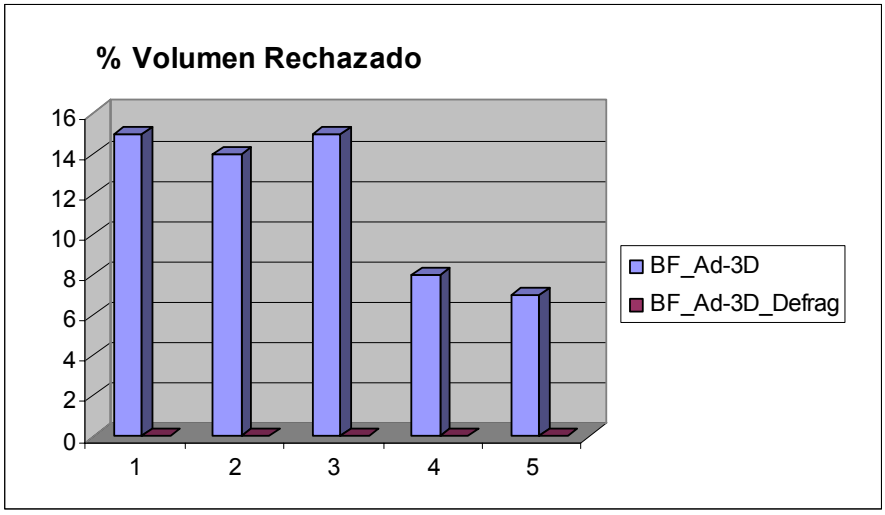


Figura 7.10. Volumen de cálculo rechazado.

La figura 7.11 muestra el nivel de ocupación de la FPGA donde se vuelve a repetir el mismo comportamiento entre las dos heurísticas BF_Ad-3D y BF_Ad-3D_Defrag.

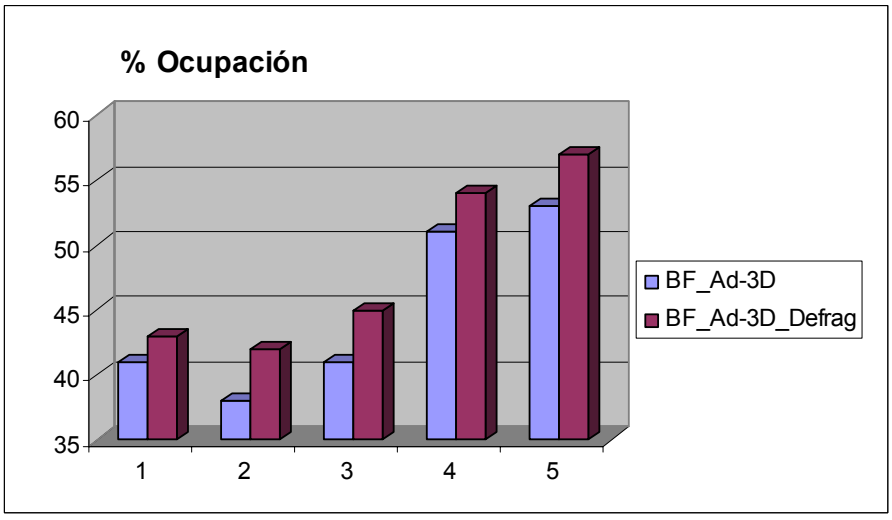


Figura 7.11. Nivel de ocupación en FPGA.

Ambas figuras muestran que cuando se aplican las técnicas de fragmentación se reduce de manera significativa el volumen de cálculo rechazado y el nivel de ocupación es mayor.

Conclusiones y trabajo futuro

En este trabajo de investigación se ha planteado el problema de la gestión de HWDR y se han revisado los subproblemas más importantes que hay que resolver, identificando de manera específica la problemática particular para permitir la multitarea HW. Los subproblemas para los que se han propuesto soluciones son la **gestión de información** sobre los recursos HW disponibles en cada momento**,** la **asignación** de recursos a tareas**,** la estimación de la **fragmentación** y la adopción de medidas de **defragmentación** del HW.

Para ello, se ha planteado un modelo de FPGA y tarea y se ha definido el tamaño mínimo de región reconfigurable, que hemos usado como unidad de gestión del área. También se ha propuesto la utilización de la estructura de datos basada en Listas de Vértices para la gestión del área libre, con la que se reduce el espacio de búsqueda de manera notable, ya que sólo se trabaja con los posibles vértices candidatos a ubicar la tarea. La Lista de Vértices se

ha comparado con otras estructuras de datos y se ha demostrado que tiene una calidad en la asignación de espacio libre a una nueva tarea semejante al del enfoque con MER solapados, pero manteniendo una complejidad similar a la de otros enfoques más simples y menos eficientes, basados en MER no solapados.

Para estimar el nivel de fragmentación de los recursos, se han propuesto dos métricas que realizan sus cálculos sobre la Lista de Vértices, una basada en el número y la complejidad de los huecos (con algunas limitaciones), FH, y otra basada en la cuadratura del perímetro del hueco, FQ. Estas métricas han sido comparadas con otras propuestas en la literatura y se ha demostrado que la basada en la cuadratura del perímetro es la que mejor refleja el grado de fragmentación del área libre.

Para el problema de la selección de un vértice concreto de entre todos los disponibles en la Lista de Vértices para la ubicación de cada tarea, se han presentado diversas heurísticas:

- En 2D, una heurística que utiliza como función de coste la métrica de fragmentación FQ, y otra basada en el concepto de adyacencia de área. Se ha comprobado que ambas proporcionan resultados semejantes entre sí y de mejor calidad que cuando se utiliza una técnica FF.
- En 3D, se han propuesto heurísticas que además de considerar el área disponible tienen en cuenta la dimensión temporal. La primera de ellas considera la adyacencia a lo largo del tiempo, y la segunda realiza dos búsquedas, una en el momento actual y otra cuando se produce la salida de la siguiente tarea. Es esta última heurística la que ha proporcionado mejores resultados para todos los ejemplos estudiados.

Por último, para el tratamiento del problema de la fragmentación, se han propuesto dos técnicas de defragmentación, cada una adecuada a diferentes tipos de situaciones: Defragmentación Preventiva (causada por la aparición de una isla o por un estado de alta fragmentación) y Defragmentación Urgente (cuando una tarea no encuentra una posición viable en la FPGA, aunque haya suficiente espacio libre). Se ha comprobado que las heurísticas

3D con técnicas de defragmentación proporcionan unos resultados excelentes, pudiendo ejecutar el 100% de las tareas para los ejemplos utilizados.

Como trabajo futuro, se pretende adaptar el Gestor de HWDR y el modelo de FPGA a modelos de arquitecturas heterogeneas y estructuras de FPGAs tridimensionales.

En nuestro modelo de FPGA se ha considerado una arquitectura 2D, con un sistema de reconfiguración en 2D y una organización homogénea. Sin embargo, en este modelo no se ha considerado la presencia de elementos de grano grueso, como la memoria RAM distribuida en el dispositivo.

Muchas aplicaciones reales realizan operaciones sobre conjuntos de datos extensos (por ejemplo en multimedia) donde se puede necesitar un banco de memoria externa a través del cual se puedan compartir datos entre las distintas tareas ejecutadas en la FPGA. Para tener en cuenta las necesidades de este tipo de aplicaciones y la presencia de elementos de memoria distribuida (BlockRam) en el dispositivo, se pueden diseñar heurísticas de búsqueda de ubicación que tengan en cuenta de manera simultánea los elementos de lógica reconfigurable y de memoria.

Además, si se utiliza la RAM interna como caché, se puede acelerar la ejecución de este tipo de aplicaciones, ya que permite tener ventaja por la localidad de datos espacial y temporal dentro de la misma región reconfigurable, reduciendo el tiempo de acceso de las tareas a los datos en memoria externa compartida. Para agregar esta funcionalidad, nuestro gestor de Hw debería considerar y gestionar no sólo las regiones reconfigurables, sino también sus bloques de memoria asociada.

Por otra parte, como se adelantó en el capítulo 7, se podría utilizar esta memoria también para el proceso de defragmentación. Para poder interrumpir la ejecución de las tareas sin tener que reiniciarlas luego de nuevo, se necesitaría intercalar en determinados puntos de su ejecución unos puntos de chequeo (checkpoints) para salvar el estado en la memoria y después de la reubicación, la ejecución podría reanudarse a partir de ese punto.

Una segunda línea de trabajo es la extensión de las técnicas de ubicación y defragmentación para multitarea HW en 2D, presentadas en este trabajo de investigación, a arquitecturas de FPGAs en 3D.

En el anexo 2, sección 2.3 se presentan algunas alternativas actuales que se han propuesto para la fabricación de FPGAs tridimensionales. Basándonos en dichas alternativas podemos concluir que plantear la gestión de recursos HW en 3D no es tan especulativo, ya que incluso en la actualidad se dispone de algunos prototipos de FPGAs 3D, aunque muy limitados, como la fabricada por la Universidad de Cornell [FLPM06], o el prototipo desarrollado por Tezzaron Semiconductor [Tezz08].

En este nuevo entorno de FPGAs 3D, pensamos proponer la extensión de las técnicas presentadas en 2D hacia 3D: las tareas serían prismas rectangulares, en lugar de los rectángulos considerados en 2D, y la gestión del espacio disponible se realizaría con un CLV tridimensional.

La métrica FQ puede adaptarse con facilidad para 3D, y estimaría cuánto se aproxima la forma del área libre en 3D a la del hueco ideal consistente en un cubo perfecto, algo que podríamos denominar “cubicidad” del hueco libre. De igual modo que en 2D, este cubo se considera que tiene la forma más adecuada para acomodar las futuras tareas.

Finalmente, las heurísticas de selección de vértices presentadas en este trabajo de investigación, basadas en área (fragmentación y adyacencia 2D) y en área y tiempo (adyacencia espacio-temporal y anticipativa 3D) podrían adaptarse igualmente a esta nueva arquitectura de FPGAs en 3D.

Publicaciones generadas

[RSMM03] S. Roman, J. Septién, H. Mecha, D. Mozos, J. Tabero, "Partition-based Algorithm for Efficient 2D HW Multitasking". Euromicro Digital System Design Conference, 2003.

[SMMT06] J. Septién, H. Mecha, D. Mozos, J. Tabero, "2D Defragmentation Heuristics for Hardware Multitasking on Reconfigurable Devices", Proc. of 13th Reconfigurable Architectures Workshop, RAW'06, Grecia, April 2006.

[SMMT08] J. Septién, D. Mozos, H. Mecha, J. Tabero and M. A. García de Dios, "Perimeter Quadrature-based Metric for Estimating FPGA Fragmentation in 2D HW Multitasking", Proc. of 15th Reconfigurable Architectures Workshop, RAW'08, 2008.

[TSMM03] J. Tabero, J. Septién, H. Mecha, D. Mozos, S. Roman, "Efficient hardware multitasking through space multiplexing in 2D RTR FPGAs". Euromicro Digital System Design Conference, 2003.

[TSMM03b] J. Tabero, J. Septién, H. Mecha, D. Mozos, "Gestión de Hardware 2D Multitarea en FPGAs Dinámicamente Reconfigurables Basado en Listas de Vértices". JCRA 2003, pag. 55-62, Madrid, España, Septiembre, 2003.

[TSMM03c] J. Tabero J. Septién, H. Mecha, D. Mozos, "A vertex-list approach to 2D HW multitasking management in RTR FPGAs", Conf. On Design of Circuits and Integrated Systems (DCIS), pag. 545-550, 2003.

[TSMM04] J. Tabero, J. Septién, H. Mecha, D. Mozos, "A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management". FPL 2004, Belgium. publicados en Springer Verlag Lecture Notes on Computer Science, vol. 3203, pag. 1178-1179. 2004.

[TSMM06] J. Tabero J. Septién, H. Mecha, D. Mozos, "Task Placement Heuristic Based on 3D-Adjacency and Look-Ahead in Reconfigurable Systems", Proc. of the 11th Asia and South Pacific Design Automation Conference, AspDAC, Japón 2006.

[TSMM08] J. Tabero, J. Septién, H. Mecha, D. Mozos, "Allocation Heuristics and Defragmentation Measures for Reconfigurable Systems Management", INTEGRATION, the VLSI journal 41 282 , pag. 281–296, 2008.

Referencias

[ABBT04] A. Ahmadinia, C. Bobda, M. Bednara, J. Teich, “A new approach for on-line placement on reconfigurable devices” 11th. Reconfigurable Architectures Workshop (RAW), Proc. Int’l Parallel and Distributed Processing Symp (IPDPS), USA, 2004.

[AFGM05] C. Ababei, Y. Feng, B. Goplen, H. Mogal, T. Zhang, K. Bazargan, S. Sapatnekar, "Placement and routing in 3D integrated circuits", IEEE Design & Test of Computers, 22 (6), pag. 520- 531, 2005.

[AGSU87] A. Aggarwal y S. Suri, “Fast algorithms for computing the largest empty rectangle”. Proceeedings of the third annual symposium on Computational geometry, pag. 278-290, 1987.

[AhBT03] A. Ahmadinia, C. Bobda, J. Teich, “Temporal task clustering for online placement on reconfigurable hardware”, Field-Programmable Technology (FPT), 2003. Proceedings IEEE International Conference, pag. 359-362, 2003.

[Atme08] www.atmel.com. AT94KAL “Series Field Programmable System Level Integrated Circuit” Data Sheet, revisión I, 2008.

[BaKS00] K. Bazargan, R. Kastner, M. Sarrafzadeh, “Fast template placement for reconfigurable computing systems”, IEEE Design and Test of Computers, 17 (1), pag. 68–83, 2000.

[BAMT05] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, J. van

der Veen, **"**DyNoC: A dynamic infrastructure for communication in dynamically reconfugurable devices**".** Proceedings of the Field Programable Logic Conference (FPL), pag. 153- 158, 2005.

[BeRo97] V. Betz, J. Rose, "VPR: A new packing placement and routing tool for FPGA research," in Field-Programmable Logic and Applications, pag. 213–222, 1997.

[BKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. "*Computational Geometry*". Berlín, Springer Verlag, 1997.

[BoAh05] C. Bobda, A. Ahmadinia, "Dynamic interconnection of reconfigurable modules on reconfigurable devices," IEEE Design & Test of Computers, 22 (5), pag. 443- 451, 2005.

[BMKA04] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich, "A dynamic noc approach for communication in reconfigurable devices," Proceedings of International Conference on Field-Programmable Logic and Applications (FPL), ser. Lecture Notes in Computer Science (LNCS), Volumen 3203. Belgica, Springer, pag. 1032–1036, 2004.

[BoSr00] A. Boulis, M. Srivastava, "System design of Active Basestations on dynamically reconfigurable hardware", IEEE/ACM Design Automation Conference DAC'2000.

[BrDi01] G. Brebner, O. Diessel. "Chip-based reconfigurable task management". In International Conference on Field-Programmable Logic and Applications, (FPL 2001), pag. 182 -191, 2001.

[Breb96] G. Brebner, "A virtual hardware operating system for the Xilinx XC6200". Proceedings of the 6th Internacional Workshop on Field Programable Logic (FPL´96), pag. 327-336, 1996.

[CCKH00] K. Compton, J. Cooley, S. Knol, S. Hauck, "Configuration Relocation and Defragmentation for Reconfigurable Computing", IEEE Symposium on Field-Programmable Custom Computing Machines, 2000.

[CDHL00] Y. Cui, X. Duan, J. Hu, and C. Lieber, "Doping and electrical transport in silicon nanowires," Journal of Physical Chemistry B, Volumen 104, no. 22, pag. 5213–5216, 2000.

[Chen03] Y. Chen , “Nanoscale molecular-switch devices fabricated by imprint lithography” in Applied Physics Letters 82, no. 10, pag. 1610-1612, 2003.

[CLCK02] K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck, “Configuration relocation and defragmentation for run-time reconfigurable computing”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN:1063-8210, Volumen 10, Issue 3, pag. 209-220, 2002.

[CoAA01] P. Collins, M. Arnold, and Ph. Avouris, “Engineering Carbon Nanotubes and Nanotube Circuits Using Electrical Breakdown,” Science, Volumen 292, pag. 706, 2001.

[CoCW02] E. Coffman, J. Csirik, G. Woeginger, “*Approximate solutions to Bin-Paking problems*”, in: P. Pardalos, M. Resende (Eds.), Handbook of Applied Optimization, Oxford University Press, 2002.

[CoHa02] K. Compton, S. Hauck, “Reconfigurable computing: a survey of systems and software”, ACM Computing Surveys, 34 (2), pag. 171-210, 2002.

[DaPr01] A. Dandalis, V. Prasanna, “Configuration Compression for FPGA-based embedded systems”, ACM/SIGDA International Symposium on FPGAs, 2001.

[DaPr05] A. Dandalis, V. Prasanna, “Configuration compression for FPGA-based embedded systems”, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on Volumen 13, Issue 12, pag. 1394-1398, Dec. 2005.

[DaVV99] J. Dambre, H. Van Marck, J. Van Campenhout, “Quantifying the performance of optoelectronic FPGA´s: the impact of optical interconnect latency”, 1999.

[DEMS00] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, B. Schmidt, “Dynamic scheduling of tasks on partially reconfigurable FPGAs”. IEE Proc.-Computer Digital Technology, Volumen 147, No. 3, pag. 181-188, May 2000.

[DeWi04] A. Dehon, M. J. Wilson, “Nanowire-Based Sublithographic Programmable Logic Arrays,” in Proc. Of International Symposium on Field Programmable Gate Arrays, 2004.

[DiEl01] O. F. Diessel, H. Elgindy, "On Dynamic Task Scheduling for FPGA-based Systems", International Journal of Foundations of Computer Science, IJFCS'01, Volumen 12, No. 5, pag. 645-669, 2001.

[Dies98] O. F. Diessel, "*On Scheduling Dynamic FPGA Reconfigurations*", Ph D. Thesis, 1998.

[DiWi99] O. F. Diessel, G. Wigley, "Opportunities for Operating Systems Research in Reconfigurable Computing", Technical report ACRC-99-018. Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, 1999.

[EGLM03] J. Edmonds, J. Gryz, D. Liang, R. J. Miller. "Mining for Empty Spaces in Large Data Sets". Theoretical Computer Science, 3(296), pag. 435–452, 2003.

[EjDe05] A. Ejnioui, R. F. DeMara, "Area Reclamation Metrics for SRAM-based Reconfigurable Device", in the Proceedings of The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05), Las Vegas, USA, 2005.

[FLPM06] D. Fang, C. LaFrieda, S. Peng, R. Manohar "A Three-Tier Asynchronous FPGA" (invited paper) 23rd VLSI/ULSI Multilevel Interconnection Conference (VMIC), 2006.

[GaVI05] A. Gayasen, N. Vijaykrishnan, M.J. Irwin, "Exploring Technology Alternatives for Nano-escale FPGA Interconnects", IEEE/ACM Design Automation Conference DAC'2005, pag. 921-926, 2005.

[GoBu01] S.C. Goldstein, M. Budiu, "Nanofabrics: Spatial computing using molecular electronics", In Proceedings of the International Symposium on Computer Architecture (ISCA 2001), 2001.

[Gold05] S.C. Goldstein, "The impact of the nanoescala on Computing Systems" in Pro. IEEE 2005.

[Golu80] M. C. Golumbic, "Algorithmic graph theory and perfect graphs". New York: Academic Press, 1980.

[GoRo02] S. Goldstein, D. Rosewater, "Digital Logic Using Molecular Electronics," in Int'l Solid State Circuits Conference, 2002.

[GSBC00] S.C. Goldstein, H.Schmit, M.Budiu, S.Cadambi, M.Moe, Taylor, R.R. Computer . Volumen 33, Issue 4, pag. 70-77, 2000.

[Hand04] M. Handa, **"**Online Placement and Scheduling Algorithms and Methodologies for Reconfigurable Computing Systems". PhD Engineering: Computer Science & Engineering, University of Cincinnati, 2004..

[Hart01] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionay Retrospective". Proceedings of the conference on Design, automation and test in Europe (DATE), pag. 642 – 649, 2001.

[Hauc98] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Processors", Proceedings of the IEEE Symposium on FPGAs, pag. 65- 74, 1998.

[HaVe04] M. Handa, R. Vemuri, **"**An efficient algorithm for finding empty space for online FPGA placement". Design Automation Conference, Proceedings, Volumen 41, pag. 960-965, 2004.

[HaVe04b] M. Handa, R. Vemuri, "Area Fragmentation in Reconfigurable Operating Systems", Engineering of Reconfigurable Systems and Algorithms (ERSA'04), Las Vegas, USA, 2004.

[HUWB04] M. Hübner, M. Ullmann, F. Weissel, J. Becker, "Real-time configuration code decompression for dynamic FPGA self-reconfiguration", 18th IEEE International Parallel and Distributed Processing Symposium, pag. 138, 2004.

[ITRS05] "*International Technology Roadmap for Semiconductors*", http://www.itrs.net/.

[KAKS97] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multi-level hypergraph partitioning: Applications in VLSI design," Proceedings of the ACM/IEEE Design Automation Conference, pag. 526–529, 1997.

[KaPo05] H. Kalte, M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems". Field Programmable Logic and Applications, International Conference, pag. 223–228, 2005.

[KaPo06] H. Kalte, M. Porrmann. “REPLICA2Pro: Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs”. In Proc. of the ACM International Conference on Computing Frontiers, 2006.

[KKKP04] H. Kalte, M. Koester, B. Kettelhoit, M. Porrmann, U. Rückert. “A comparative study on system approaches for partially reconfigurable architectures”. In Proc. of the Int. Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA ’04), CSREA Press, pag. 70-76, 2004.

[KLPR05] H. Kalte, G. Lee, M. Porrmann, U. Rückert. “REPLICA: A bitstream manipulation filter for module relocation in partial reconfigurable systems”. In Proc. of the 19th International Parallel and Distributed Processing Symposium, 2005.

[KoDi06] S. Koh, O. Diessel. “COMMA: A Communications Methodology for Dynamic Module Reconfiguration in FPGAs”. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’06), pag. 273-274, 2006.

[KoPK05] M. Koester, M. Porrmann, H. Kalte, “Task Placement for Heterogeneous Reconfigurable Architectures”. in Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference, pag. 43-50, 2005.

[KoPK06] M. Koester, M. Porrmann, H. Kalte, “Relocation and defragmentation for heterogeneous reconfigurable systems”. In Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '06), Las Vegas, USA. CSREA Press, 2006.

[KuRo07] I. Kuon, J. Rose, "Measuring the Gap Between FPGAs and ASICs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) , Volumen 26 Nº 2, pag. 203-215, 2007.

[Maki00] T. Makimoto, “The Rising Wave of Field-Programmability”, Proc. PL2000, Aug. 27-30, 2000; LNCS, Springer-Verlag 2000.

[MBVL02] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, R. Lauwereins. "*Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs*". Proceedings of the Field Programable Logic Conference (FPL), pag.

795-805, 2002.

[McFL03] M. C. McAlpine, R. S. Friedman, C. M. Lieber, "Nanoimprint Lithography for Hybrid Plastic Electronics," Nano Lett. 3, pag. 443-445, 2003.

[MeLJ98] P. Merino, J.C. López, M. Jacome, "A hardware operating system for dynamically reconfiguration of FPGAs", en Field-Programmable Logic and Applications, FPL'98, Springer-Verlag, pag. 431–435, 1998.

[MNCV03] J.Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins, "Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip". In Proceedings of DATE 2003 Conference, Alemania, 2003.

[Möhr85] R. H. Möhring, "Algorithmic aspects of comparability graphs and interval graphs," in Graphs and Order, I. Rival, Ed. D. Reidel Publishing Company, Dordrecht, pag. 41–101, 1985.

[NeGa95] M. Nelson, J. L. Gailly, "*The data compression book*" (2nd ed.), MIS:Press, New York, 1995.

[NoBa01] J. Noguera, R. Badia, "A HW/SW Partitioning Algorithm for Dynamically Reconfigurable Architectures", DATE'2001.

[RDCR03] A. Rahman, S. Das, A. P. Chandrakasan, R. Reif. "Wiring requirement and three-dimensional integration technology for field programmable gate arrays". In IEEE Transactions on Very Large Sale Integration (VLSI) Systems, Volumen 11, pag. 44–54, 2003.

[RCGM08] J. Resano, J. Clemente, C. González, D. Mozos, F. Catthoor , "Efficiently scheduling run-time reconfigurations". Transactions on Design Automation of Electronic Systems, Volumen 13 Nº 4, pag. 58-69, Septiembre 2008.

[RMMS08] S. Román, H. Mecha, D. Mozos, J. Septién "Constant complexity scheduling for hardware multitasking in two dimensional reconfigurable field-programmable gate arrays" IET Computers & Digital Techniques Volumen 2, Nº 6 pag. 401-412, 2008.

[RSMM03] S. Román, J. Septién, H. Mecha, D. Mozos, J. Tabero, “Partition-based algorithm for efficient 2D HW multitasking”. EUROMICRO WIP, pag. 26-27, 2003.

[SBBA06] P. Sedcole, B. Blodget, T. Becker, J. Anderson, P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs". IEE Proceedings: Computers and Digital Techniques, Volumen 153 Nº 3, pag. 157-164, 2006.

[Slea80] D. Sleator, “A 2.5 times optimal algorithm for bin packing in two dimensions”, Information Processing Letters 10, pag. 37- 40, 1980.

[StWP04] C. Steiger, H. Walder, M. Plazner. “Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks”, IEEE Transactions on Computers, Volumen 53, Nº 11, pag. 1393-1407, 2004.

[TCJW97] S. Trimberger, D. Carberry, A. Johnson, J. Wong. “A time-multiplexed FPGA”. Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97), pag. 22-29, 1997.

[Tezz08] http://www.tezzaron.com/about/PhotoAlbum/Products/.

[TSMM03] J. Tabero J. Septién, H. Mecha, D. Mozos, “A vertex-list approach to 2D HW multitasking management in RTR FPGAs”, Conf. On Design of Circuits and Integrated Systems (DCIS), pag. 545-550, 2003.

[TSMM04] J. Tabero, J. Septién, H. Mecha, D. Mozos, “A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management*”*. Proceedings of the Field Programable Logic Conference (FPL’04), pag. 241-250, 2004.

[TVHH02] J. M. Tour, W. L. Van Zandt, C. P. Husband, S. M. Husband, L. S. Wilson, P. D. Franzon, D. P. Nackashi, “Nanocell Logic Gates for Molecular Computing,” IEEE Transactions on Nanotechnology 2002, pag. 100-109, 2002.

[UHGB04] M. Ullmann, M. Hübner, B. Grimm, J. Becker, “An FPGA Run-Time System for Dynamical On-Demand”. Reconfiguration. 11th Reconfigurable Architectures Workshop (RAW), USA, 2004.

[VFMA05] J. C. Van der Veen, S. Feteke, M. Majer, A. Ahmadinia, C. Bobda, F. Hanning, J. Teich, “ Defragmenting the Module Layout of a Partially

Reconfigurable Device," Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'05).

[WaPl02] H. Walder, M. Platzner, "Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform", Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'02), pag. 24-30, 2002.

[WaPl03] H. Walder, M. Platzner, "Online Scheduling for Block-partitioned Reconfigurable Devices". Proceedings of the Design Automation & Test in Europe Conference (DATE'03), pag. 10290-10295, 2003.

[WaPl03b] H. Walder, M. Platzner, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations". Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'03), 2003.

[WaSP03] H. Walder, C. Steiger, M. Platzner, "Fast online task placement on FPGAs: free space partitioning and 2D-Hashing" 10th Reconfigurable Architectures Workshop (RAW), Proc. Int Parallel and Distributed Processing Symposium (IPDPS), Francia, 2003.

[WiKe01] G. B. Wigley, D. A. Kearney. "The First Real Operating System for Reconfigurable Computers". Proceedings of the 6th Australasian Computer Systems Architecture Conference, pag. 130-137, 2001.

[WiKe02] G. B. Wigley, D. A. Kearney. "Research Issues in Operating Systems for Reconfigurable Computing", In proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms (ERSA'02), pag. 10-16, 2002.

[WiKe02b] G. Wigley, D. Kearney. "The Management of Applications for Reconfigurable Computing Using an Operating System". In Proceedings of the seventh AsiaPacific conference on Computer systems architecture, Volumen 6, pag. 73--81, 2002.

[Xili96] XC6200: Advance Product Specification, Xilinx, Inc., San Jose, CA: 1996.

[Xili98] www.xilinx.com/support/documentation/data_sheets/XC3000 series: Product Specification, Xilinx, Inc., San Jose, CA: 1998.

[Xili99] www.xilinx.com, “XC4000, Product Specification”, DS015 (v1.3) October 18, 1999 - Product Specification.

[Xili04] www.xilinx.com. **“**Virtex Series Configuration Architecture User Guide”. XAPP151 (v1.7) October 20, 2004

[Xili02] www.xilinx.com. “Virtex™ 2.5 V Field Programmable Gate Arrays- Product Specification”. DS003-1 (v2.5) April 2, 2001.

[Xili04b] Xilinx Inc. “Two flows for partial reconfiguration: module based or difference based”. Xilinx Application Note XAPP290 Sep., 2004.

[Xili07] www.xilinx.com. ”Virtex-II Pro and Virtex-II Pro X FPGA User Guide” UG012 (v4.2) 5 November 2007.

[Xili07b] www.xilinx.com.”Virtex-4 User Guide”. UG070 (v2.3) August 10, 2007.

[Xili07c] www.xilinx.com.”Virtex-4 Configuration Guide”. UG071 (v1.9) October 1, 2007.

[Xili07d] www.xilinx.com.”Virtex Series FPGAs Product Selection Guide”. 2007.

[Xili08] www.xilinx.com.” Virtex-5 User Guide”. UG190 (v3.3) February 5, 2008.

[Xili08b] www.xilinx.com.”Virtex-5 Configuration Guide”. UG191 (v2.7) February 1, 2008.

Apéndice 1:

Arquitecturas de FPGAs de Xilinx

En este apéndice se revisan en detalle las arquitecturas más relevantes de FPGAs de uno de los principales fabricantes como es Xilinx. Se destacan los aspectos más importantes y que más relación tienen con el trabajo presentado, desde el punto de vista de la gestión dinámica de dispositivos reconfigurables, como son las distintas características arquitectónicas, el tamaño, el sistema de reconfiguración, las nuevas funcionalidades y la gestión del reloj. Además se presentarán algunas soluciones propuestas por grupos de investigación para permitir la comunicación de tareas.

Nota: Todas las figuras correspondientes a las arquitecturas de Xilinx son propiedad de Xilinx.

A1.1 Principales arquitecturas de FPGAs de Xilinx

A. Xilinx XC6200

Aparece en 1996 y se trata de una arquitectura de grano fino, de estilo mar de celdas que permite reconfiguración parcial a nivel de celda. Esta familia tiene una estructura simple, simétrica, jerárquica y regular. No tuvo el éxito comercial esperado y Xilinx la dejó de fabricar pronto porque no había herramientas que permitieran explotar todas sus posibilidades, ya que su flujo de diseño estaba basado principalmente en la herramienta XACTStep series6000 y no se disponía de una herramienta específica para realizar la reconfiguración dinámica. La única herramienta disponible era Jbits que permitía únicamente pequeñas manipulaciones de bits del bitstream. Sin embargo, esta familia es muy importante a nivel de investigación ya que es la primera que permite la reconfiguración parcial.

Arquitectura. Desde el punto de vista de organización física y lógica esta familia es muy jerárquica. En el nivel bajo de esta jerarquía se encuentra un array de gran tamaño de celdas simples (hasta 24.000 puertas en la XC6216), como se puede ver en la figura A1.1, donde se muestran en (a) la estructura de interconexión entre celdas vecinas y en (b) un nivel superior de la jerarquía entre bloques de 4*4 celdas vecinas. Cada nivel de jerarquía se escala por un factor de 4*4, donde las conexiones de mayor longitud en cada nivel se denominan *FastLanes*. En cada bloque existen una serie de switches distribuidos por la periferia, que proporcionan las conexiones entre los distintos niveles en la misma posición del array.

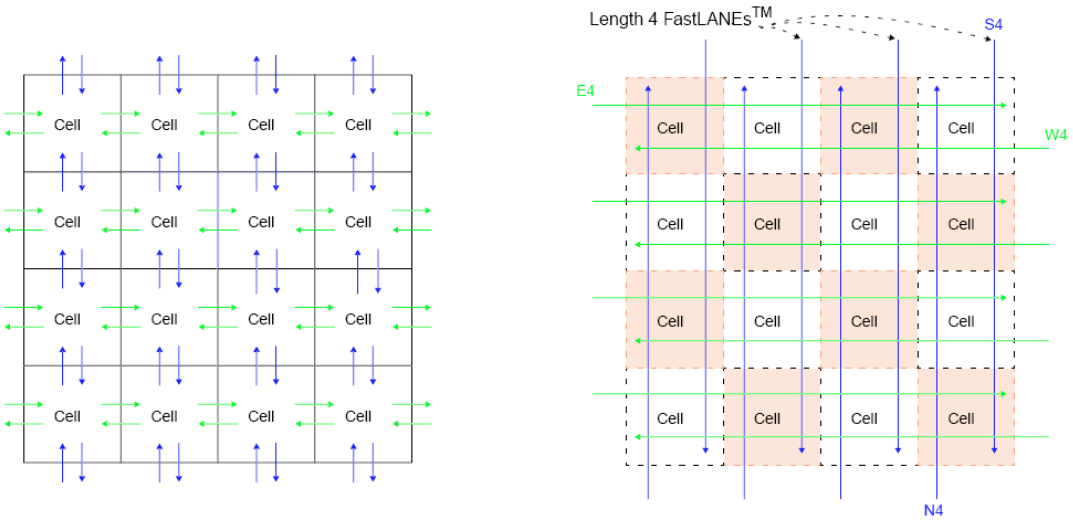


Figura A1.1. Estructura de interconexión de la XC6200, (a) entre celdas vecinas y (b) entre bloques de 4*4 celdas.

En la XC6216, los bloques más grandes, que son de 64*64 celdas, proporcionan las conexiones de longitud máxima (*Chip-Length*) y en arquitecturas de mayor tamaño se sigue este proceso con más niveles. El beneficio que se obtiene con la jerarquía de interconexionado es que los retardos se escalan de forma logarítmica en lugar de lineal como en las primeras generaciones de FPGAs. Hay una línea adicional de Magic para interconexionar cada celda con las esquinas de los bloques de 4*4. Las celdas básicas en el array tienen entradas desde los hilos de longitud 4 asociados con bloques de 4*4 celdas, además de con las celdas vecinas más cercanas. Además hay 4 líneas globales de bajo *skew* para reset y relojes. Estas señales globales entran a través de pines de entrada dedicados y se distribuyen por el dispositivo en un patrón especial con forma de “H” de bajo *skew*. Cada grupo alineado verticalmente (Norte a Sur) de cuatro celdas dentro de un bloque de 4*4 está sincronizado por su propia fuente de reloj. La configuración se determina mediante celdas de SRAM de 6 transistores direccionables.

Configuración. La familia XC6200 dispone de un interfaz totalmente paralelo conocido como *FastMap* (bus de 32bits) que permite que las celdas lógicas (Unidades funcionales y registros) y la memoria de configuración se accedan como memoria convencional. Los bits de configuración y los de

estado se mapean en zonas distintas de memoria y la señal GClk se utiliza como reloj.

Para permitir transferencias con gran ancho de banda entre un procesador y las celdas internas se pueden transferir palabras completas de hasta 32 bits en un ciclo de memoria. Por esta razón el acceso en modo bit está mapeado en una región separada del espacio de direcciones del dispositivo.

La figura A1.2 muestra el mapa del espacio de direcciones: hay 64 columnas de celdas y una dirección de columna de 6-bit selecciona una columna en particular (para una XC6216). En el modo de escritura de configuración, el campo fila selecciona una fila, y el offset una parte concreta de la celda.

En la Lectura/Escritura (L/E) del estado se accede simultáneamente al registro de estado de hasta 32 celdas de la misma columna. Este tipo de acceso impone una limitación en la asignación de registros dentro de un diseño cuando se va a acceder en modo palabra, ya que para acceder a todos los bits del registro estos deben estar colocados en la misma columna de celdas dentro del array.

Las transferencias de datos pueden ser de 8, 16 o 32 bits, incluso cuando los bits están distribuidos sobre una columna de celdas. Si el patrón que se va a escribir se repite, se puede escribir simultáneamente en el registro de todas las celdas de una columna. Las Lecturas de estado se pueden hacer de hasta 32 bits de una misma columna (en orden correlativo, no necesariamente consecutivas).

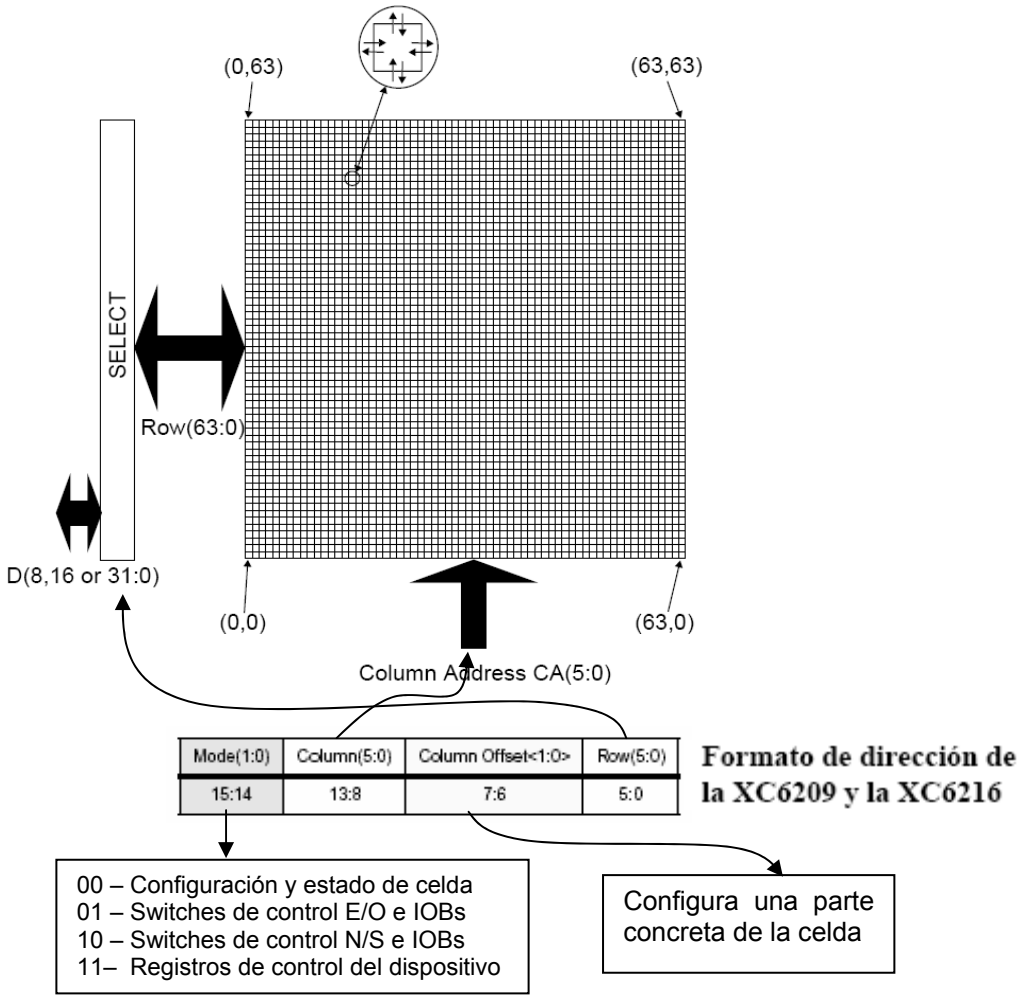


Figura A1.2. Direccionamiento de E/S.

En el modo 00 (modo celda) los 6 bits de fila y columna direccionan una celda concreta, y dentro de la celda, los 2 bit de *Column Offset* determinan el byte de RAM de configuración al que se accede. Cada celda tiene 3 bytes de configuración separados y 1 registro de 1 bit. En el modo 01 se direccionan los switches que controlan las lineas *FastLane* Este y Oeste y los IOBs situados a lo largo de los bordes Este y Oeste. En el modo 10 se direccionan los switches que controlan las lineas *FastLane* Norte y Sur y los IOBs situados a lo largo de los bordes Norte y Sur.

La figura A1.3 muestra el direccionamiento dentro del modo celda (mode=00) cuando *Column Offset* = 00 se direcciona con un solo byte todas las lineas de selección de los multiplexores de interconexionado con las

celdas vecinas. Los bytes 01 y 10 controlan el resto de interconexiones de la celda.

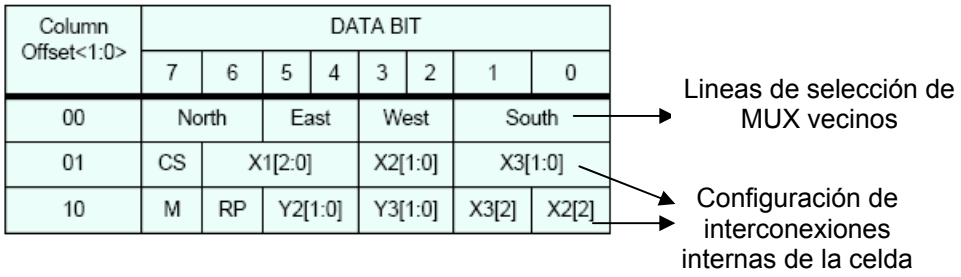


Figura A1.3. Configuración de interconexionado de la celda.

El valor de *Column Offset* =11 se usa para acceder a los bits de estado y permite sacar y meter tareas restaurando su estado. También es un mecanismo para E/S de datos: cuando se accede al estado, todos los bits que se leen son de estado. La XC6200 proporciona un mecanismo para mapear todas las salidas de las celdas de una columna en el bus externo. El Registro Map mapea las celdas de una columna sobre el bus externo (de 8, 16 ó 32 bits) seleccionando sólo aquellas que deseemos mediante un bit. Un bit a 0 indica que debe accederse a la celda de dicha fila, si hay más 0's que bits válidos, el patrón del bus se repite. En la figura A1.4 se muestra el mapeo de una columna sobre el bus externo con 8 bits.

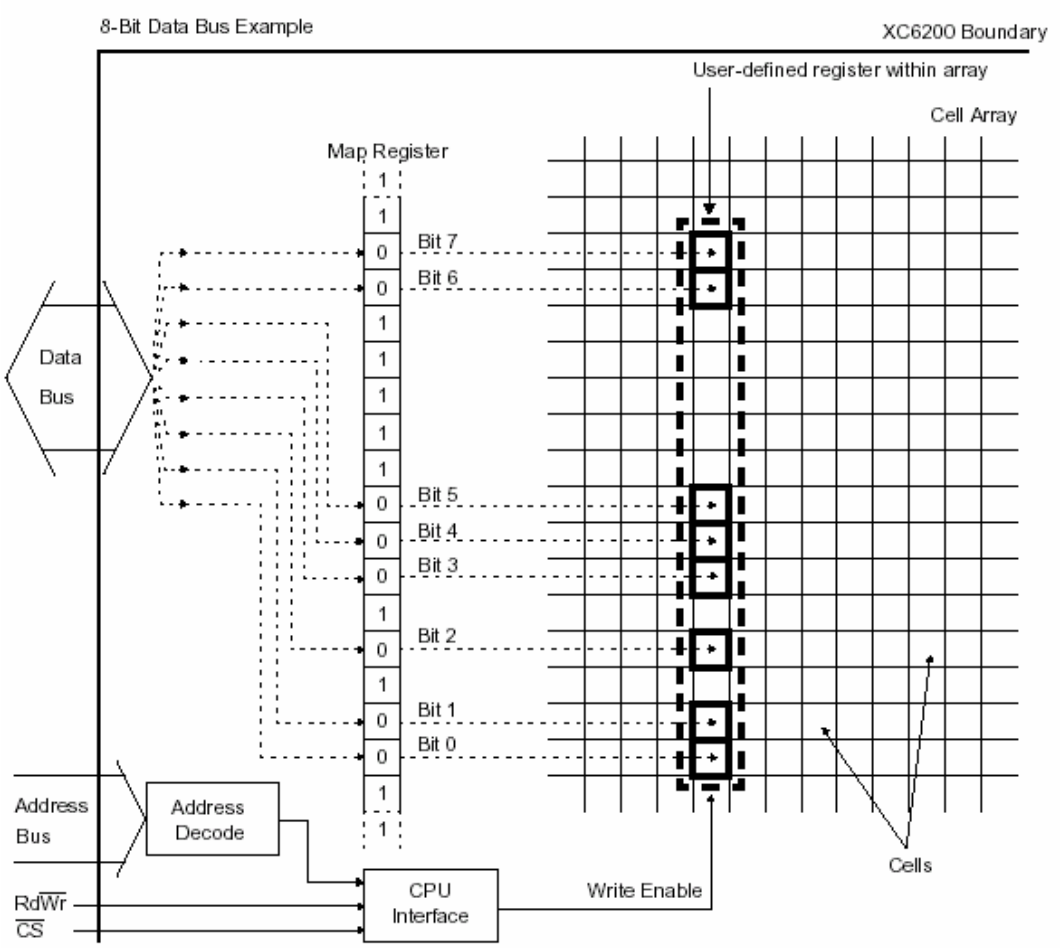


Figura A1.4. Mapeo selectivo del estado de celdas al registro interno MAP.

La XC6200 también incluye mecanismos para comprimir configuraciones: mediante el Registro de máscara y el Registro de "wildcard".

Registro de mascara: es un registro de 32 bit, situado entre el bus externo de datos y las conexiones internas de datos. Un "1" en un bit de este registro indica que el bit correspondiente del bus de datos no es relevante (ni se lee ni se escribe). Este registro se ignora durante las lecturas y escrituras de los registros de control.

Registro de "wildcard": permite acceder a varias filas o columnas simultáneamente. Un 1 en un bit de este registro indica que el bit de dirección correspondiente es irrelevante. Se desactiva en las escrituras a los registros de control y cuando se lee el estado. Existe otro registro de wildcard para columnas (también activo en escrituras de estado) que sirve para escribir en varios bancos simultáneamente.

En la figura A1.5 se muestra un ejemplo con tres tipos de accesos, en la izquierda se accede sólo a la dirección 010101, por lo que en el registro de Wildcard todos los bits están a 0. En el centro se accede simultáneamente a cuatro direcciones porque en el registro de Wildcard se ha escrito 100001, por lo que la dirección es x1010x. En la derecha se accede a ocho direcciones consecutivas porque en el registro de Wildcard se ha escrito 000111, y la dirección es 010xxx.

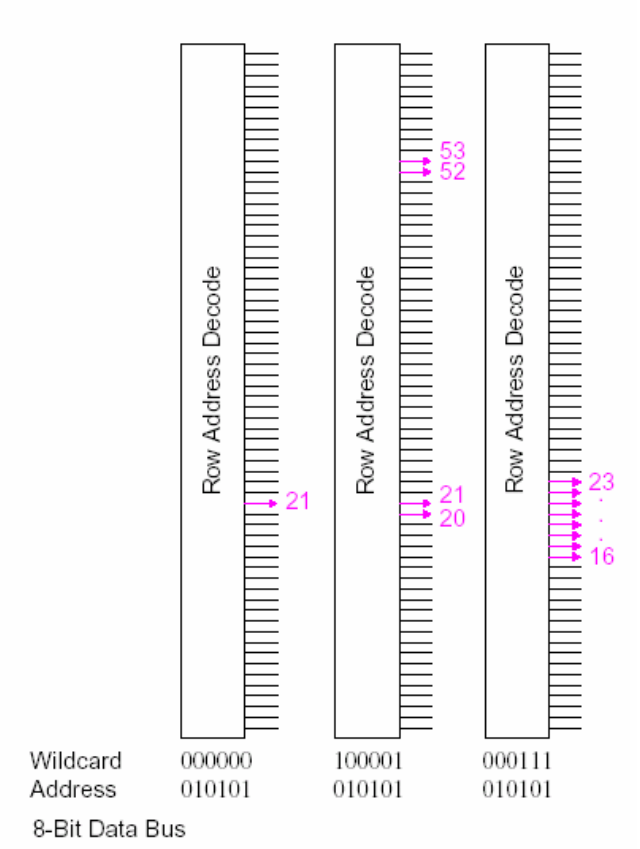


Figura A1.5. Acceso al registro Wildcard.

B. Virtex

La familia Virtex sustituye a la XC6200 y aunque mantiene algunas de sus características, no tiene la capacidad de direccionar individualmente cada celda. Para esta familia Xilinx propone dos tipos de reconfiguración parcial: basada en módulos (*Module-based*) y diferencial (*Difference-based*) en la nota de aplicación xapp290 [Xili04b]. La reconfiguración parcial basada en módulos presenta dos vertientes en función de si los módulos se comunican

entre sí (a través de *bus macro*) o no. La reconfiguración basada en diferencias se emplea cuando se realizan pequeños cambios en el diseño.

Esta familia fue presentada en 1998, y fabricada en 2.5V y tecnología de proceso de 220nm. Posteriormente la Virtex-E con tecnología de 180nm apareció en 1999 y como evolución de las anteriores aparece en 2000 la Virtex-EM, que expande las capacidades de memoria para soportar aplicaciones tales como conmutadores de red de 160Gps y video de alta definición. Está soportada por los sistemas de desarrollo ISE™ Foundation™, ISE Alliance™, and ISE BaseX™ Development Systems. Esta familia tiene densidades desde 50K puertas hasta 1 millón de puertas. La arquitectura, interconexión y complejidad de los bloques configurables cambia de manera significativa, y aparecen elementos de grano grueso. Como consecuencia del incremento en la densidad de los dispositivos, ya no es posible acceder a cada celda independientemente.

La principal característica diferenciadora es que se soporta por primera vez la reconfiguración parcial dinámica (si exceptuamos el caso de la familia XC6200).

Arquitectura. La figura A1.6 muestra la disposición de todos los componentes en la FPGA. Se puede observar que hay 2 columnas BRAM de bloques de memoria en los laterales y 4 DLLs en las esquinas, CLBs en la zona central, IOBs en la zona periferica del dispositivo y el VersaRing (para interconexionado), que se explican a continuación.

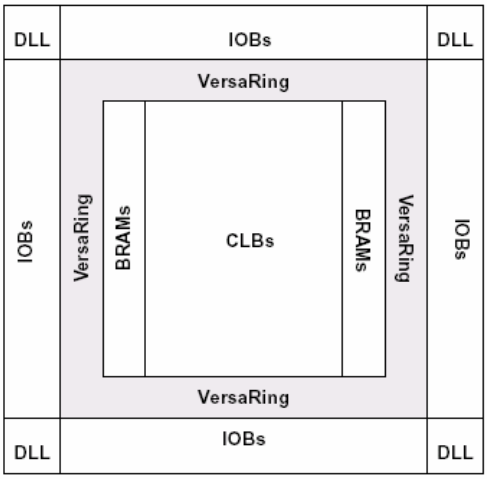


Figura A1.6. Arquitectura de Virtex.

Bloque configurable CLB. Cada CLB tiene 2 Slices similares a los que se muestran en la figura A1.7. Cada Slice tiene 2 LC (Logic Cell) y existe lógica adicional (2 puertas básicas y 2 buffer triestate) que permite contabilizar 1 CLB = 4.5 LC.

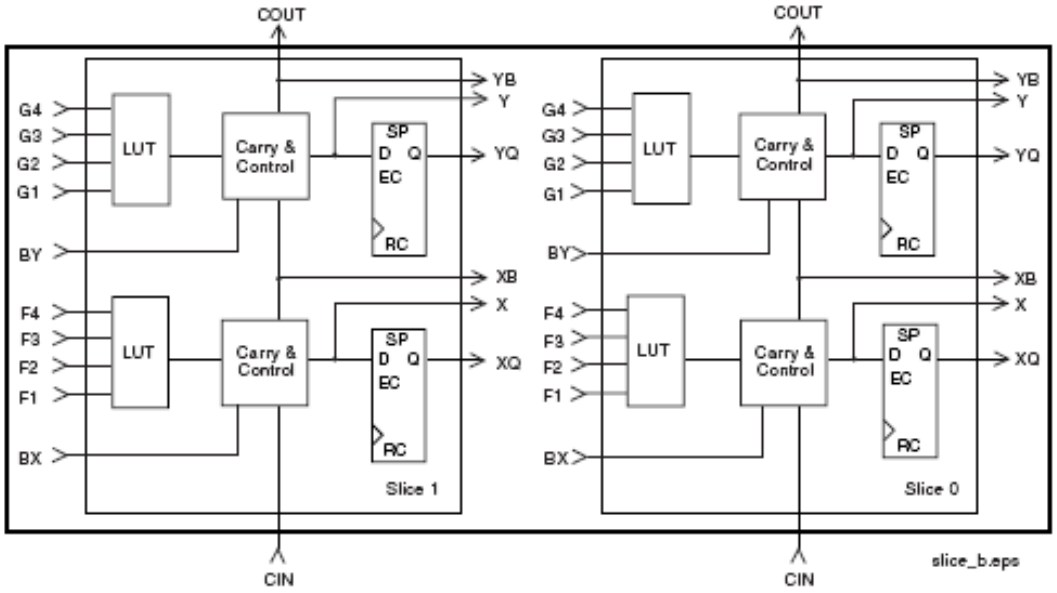


Figura A1.7. CLB de Virtex con dos Slices.

Los generadores de funciones están implementados sobre las LUTs (*Look-Up-Tables*) de 4 entradas. Además estas LUTs pueden funcionar como RAM de 16*1bit síncronas, o bien pueden combinarse 2 LUTs del mismo slice para crear una RAM síncrona de 16*2bits o 32*1bit, o bien una RAM de doble puerto de 16*1 bit. También la LUT puede funcionar como registro de desplazamiento, muy útil para captura de datos en modo ráfaga o de alta velocidad, o bien utilizarse en aplicaciones como procesamiento digital de señal.

Interconexionado. Los CLBs se conectan a través de una matriz de interconexión GRM (General routing matrix) y cada CLB se conecta a un VersaBlock que proporciona interconexionado local entre los elementos de un CLB (LUTS, Flip-flops, etc) y para conectar el CLB a la GRM. Hay líneas con *buffers* (Hex lines) en cada dirección que conectan cada GRM con otra GRM separada 6 bloques, que se pueden acceder en los extremos y en el punto medio. Además hay otras líneas largas en cada dirección y 4 líneas largas

particionables por fila conectadas a los 2 buffer triestado de cada CLB. El interfaz de E/S VersaRing proporciona interconexionado adicional alrededor del dispositivo. Los BlockRAM también se conectan a la GRM.

Gestión de reloj. Hay 4 líneas globales dedicadas, cada una conectada a un buffer global. Además dispone de 24 líneas secundarias (12 arriba y 12 abajo). Asociado a cada buffer global existe un DLL (Delay-Locked Loop) que elimina el retardo en la distribución de la señales de reloj y puede conectarse a dos líneas globales. También puede generar múltiplos y divisores de la frecuencia del reloj. En total hay 4 DLLs situados en las esquinas del dispositivo como muestra la figura A1.6.

BlockRAM. La familia Virtex incorpora varios bloques de memoria que complementan la memoria distribuida implementada sobre las LUTs (LUTRAM). Existen dos columnas de RAM en cada Virtex que se distribuyen en bloques síncronos de 4096bits (Block SelectRAM) y doble puerto. Cada bloque de 4096bits de memoria ocupa lo que 4 CLBs y la cantidad de memoria de este tipo depende del número de filas de CLBs. Aparece una jerarquía en el sistema de memoria: LUTs y BlockRAM. Añade lógica especial para acarreo, multiplicadores y encadenamiento de funciones.

IOBs. Soporta una gran variedad de estándares de señalización (16). Los 3 elementos de almacenamiento pueden funcionar como reloj o como latches, comparten el reloj, tienen señales de capacitación individuales y la señal de Set/Reset es compartida y configurable.

Los bloques de E/S se agrupan en bancos cada uno con sus entradas de Vcco y Vref. Como algunos estándares necesitan distintos Vcco y Vref, existen restricciones sobre las posibles combinaciónes de estándares en cada banco. La figura A1.8 muestra la disposición de los bancos de E/S alrededor del dispositivo. Cada borde de la FPGA se separa en dos bancos para proporcionar ocho en total.

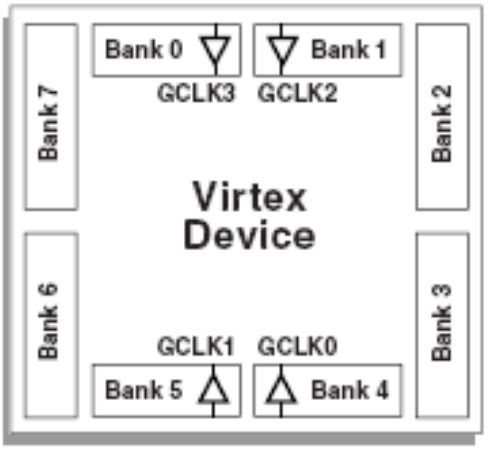


Figura A1.8. Bancos de E/S en Virtex.

Formato del stream de datos. Los dispositivos Virtex se configuran secuencialmente y los datos de configuración almacenados en la memoria de configuración se pueden releer para verificación o depuración de errores en tiempo real. Además de los datos de configuración también es posible leer los contenidos de todos los flip-flops/latches, LUTRAMs, y Block RAMs.

Arquitectura de configuración de las Virtex. Organización, estructura y direccionamiento.

En Virtex la estructura de la memoria de configuración se realiza como una matriz bidimensional. Los bits están agrupados en frames verticales (unidad mínima de configuración) de 1 bit de ancho que se extienden a la altura total del dispositivo. A su vez las frames se agrupan por columnas que pueden ser de diferentes tipos, como se muestra en la figura A1.9. Los tipos de columna pueden ser central, de CLBs, de IOBs, de interconexión de BlockRam y de contenido de BlockRam. En el interior de cada tipo de columna aparece el número de frames que la forman.

El espacio de direcciones total del dispositivo se divide en dos tipos de bloques: de RAM y de CLB. El tipo RAM es sólo para el contenido de la memoria BlockRam (no para interconexión), mientras que el tipo CLB es para el resto de columnas. A su vez los dos espacios de direccionamiento están subdivididos en direcciones principal y secundaria, por lo que la dirección

está dividida en tres partes: tipo (CLB o RAM), dirección mayor (columna) y dirección menor (número de frame dentro de la columna).

Se empieza a numerar desde la columna central y la numeración de la dirección principal depende del tipo de dispositivo, pero las direcciones pares están siempre en la mitad izquierda, mientras que las impares están en la mitad derecha, como se puede observar en la parte inferior de la figura A1.9 que corresponde a una Virtex XCV50.

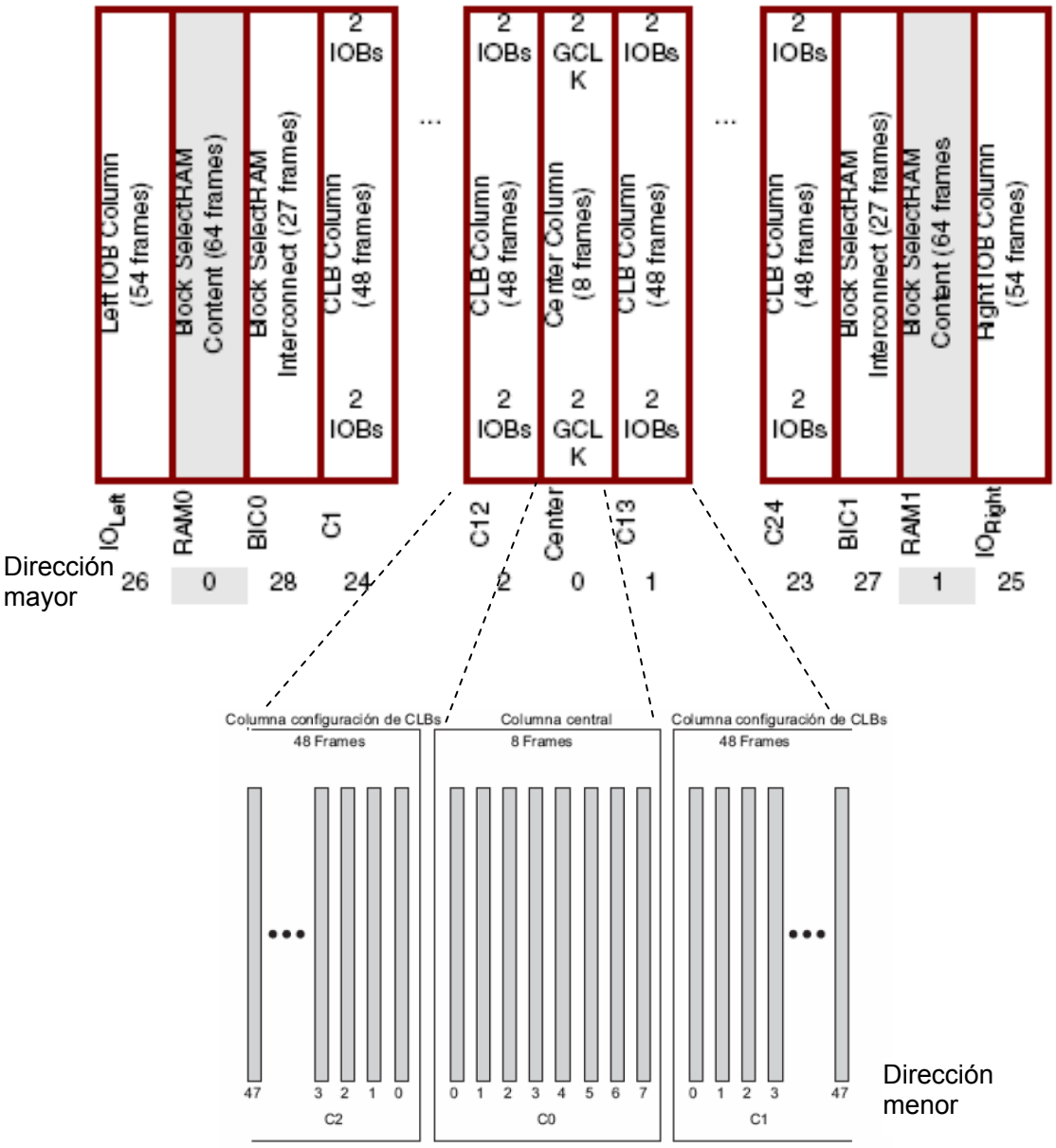


Figura A1.9. Distribución de columnas en una XCV50.

C. Virtex-II

Aparece en el año 2000 y se anuncia como la primera solución para plataforma basada en FPGA. Es una evolución de la familia Virtex y las mejoras con respecto a generaciones anteriores son el aumento de la densidad hasta 8 millones de puertas, aumento de la velocidad de reloj interno hasta 420 MHz (200MHz en Virtex-I) y de la velocidad de E/S hasta más de 840 Mb/s en LVDS (*Low Voltage Differential Signaling I/O*). Se fabrica en tecnología 150nm y aparecen encapsulados BGA para poder proporcionar mayor número de E/S.

En las Virtex-II se incrementa la memoria distribuida LUTRAM hasta 1,5 Mb y la BlockRAM hasta 3Mb, aumentando el tamaño de bloque hasta 18kb. En cuanto a E/S se incrementa el número de pines y se puede controlar la impedancia de E/S digitalmente. Se incrementa el número de controladores de la señal de reloj hasta 12, que ahora se denominan DCMs (*Digital Clock Managers*) son más complejos y añaden funciones como la posibilidad de generar relojes desplazados 90, 180 o 270 grados.

El sistema de interconexionado es jerárquico (*Active Interconnect Technology*), y se han aumentado los recursos de interconexión (por cada fila y columna existen 24 líneas largas, 120 líneas hex, 40 líneas dobles y 16 líneas directas). Aparecen multiplicadores explícitamente asociados a cada bloque de RAM. La disposición de todos los elementos comentados se muestra en la figura A1.10.

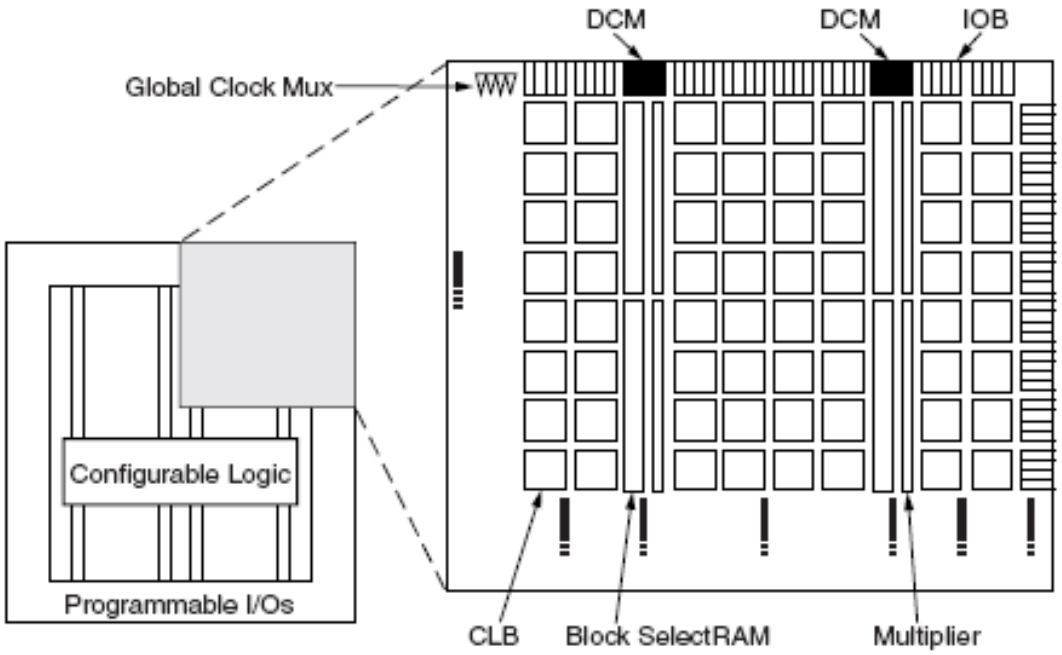


Figura A1.10. Arquitectura de Virtex-II.

IOB. Soporta una amplia variedad de estándares de E/S. Los pares diferenciales se conectan siempre a la misma matriz de conmutación. Se clasifican en bancos, y todos los elementos de un banco están conectados a un determinado VREF y VCCO.

Cada IOB incluye 6 elementos de almacenamiento configurables como latches o FF como se muestra en la figura A1.11. Se pueden configurar como bloques de entrada con Registros SDR (*Single-Data Rate*) o DDR (*Double Data Rate*), bloques de salida con SDR o DDR opcionales y un buffer tri-state y bloques bidireccionales.

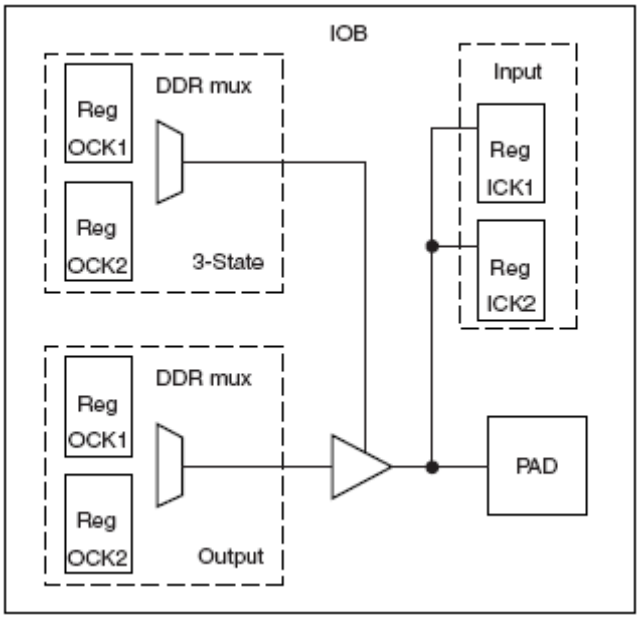


Figura A1.11. Configuración de registros en un bloque IOB.

Aparece el mecanismo DDR que puede usarse para propagar una copia del reloj, de forma que datos y reloj tengan un retardo idéntico

CLB. Cada CLB incluye 4 slices interconectados entre sí por conexiones rápidas y existe lógica e interconexionado especial para aceleración de carry (2 carrys por CLB).

Cada Slice tiene 2 generadores de funciones F y G, dos elementos de almacenamiento, puertas lógicas, multiplexores, encadenamiento de carry y encadenamiento horizontal (puerta OR). Cada LUT puede implementar memoria Select RAM (16 x 1bit) configurables en cada CLB con distintas organizaciones (Single-Port 16 x 8 bit RAM ó Dual-Port 64 x 1 bit RAM). Cada LUT también puede configurarse como registro de desplazamiento de 16 bits encadenables dentro de cada CLB

Buffer triestate. Cada uno de los 4 slices tiene acceso a los 2 buffers a través de la matriz de conmutación como muestra la figura A1.12. Además cada buffer tiene acceso alternativo a dos de las 4 líneas horizontales.

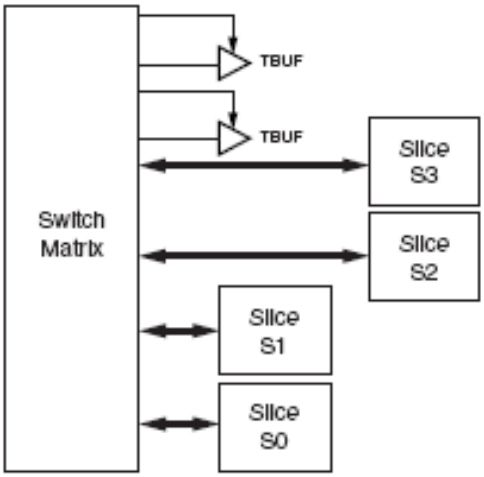


Figura A1.12. Detalle de acceso de Slices a buffers triestate.

Multiplicadores y memoria. El multiplicador y la memoria BlockRAM pueden usarse simultáneamente, aunque comparten parte del interconexionado. Existen restricciones de hasta 18 bits de ancho en el uso de la memoria cuando se usa el multiplicador. Existe un bloque de SelectRam que puede usarse como entrada al multiplicador, permitiendo la implementación de DSPs.

Interconexionado. Tiene el siguiente interconexionado dedicado: 16 lineas de relojes globales, de las que se pueden usar 8 por cuadrante, 4 líneas horizontales por fila de CLBs conectadas a los buffers tri-state, 2 líneas especiales por columna para propagación de carry, 2 líneas dedicadas por fila para propagar las salida de la OR con el slice adyacente, y 1 línea vertical para conectar los registros de desplazamiento de LUT a LUT.

La figura A1.13 muestra la disposición de las 16 lineas de reloj globales y la distribución de reloj por cuadrantes. En cada cuadrante las 8 lineas de reloj se organizan en filas, que pueden soportar hasta 16 filas de CLBs (8 arriba y 8 abajo).

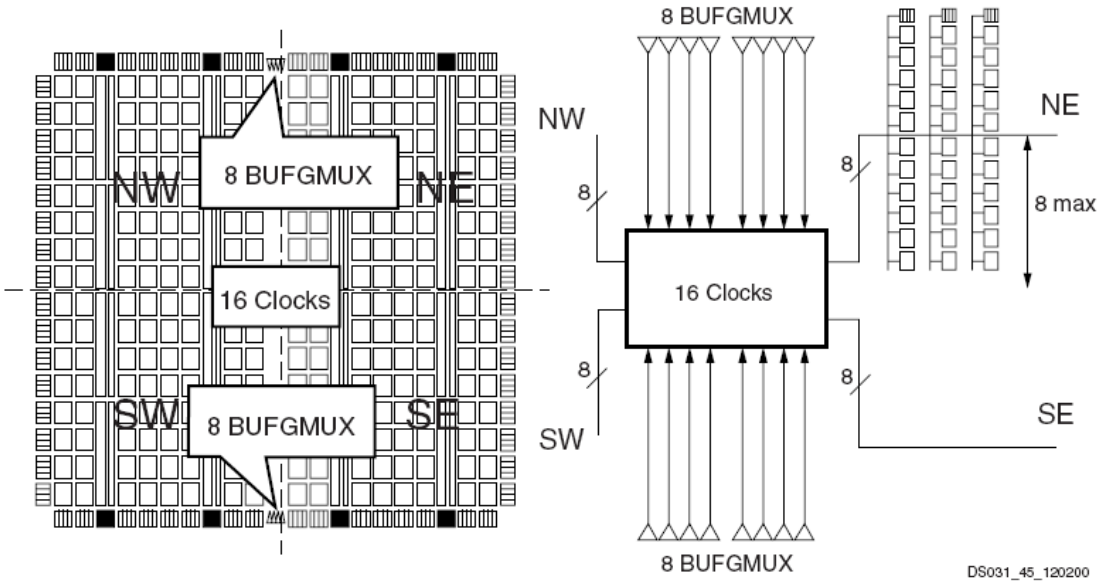


Figura A1.13. Distribución de reloj por cuadrantes en Virtex-II.

D. Virtex-II Pro y Virtex-II Pro X

Aparece en 2002 y su arquitectura está basada completamente en la Virtex-II y la mayoría de las características son idénticas. Son plataformas para diseños que estén basados en bloques predefinidos (*IP cores*) y módulos personalizados como telecomunicaciones, wireless, aplicaciones en red, video y aplicaciones DSP. Se fabrica en tecnologia de 130nm. Los modelos tienen capacidades similares desde 230k a 7 millones en Virtex-II Pro y desde 40k a 8 millones en Virtex-II de puertas equivalentes.

El interconexionado, los DCMs, los bloques de BlockRam, los multiplicadores y los CLBs están en idéntica configuración. Las principales diferencias son:

Es la primera en incorporar transmisores/receptores de alta velocidad Multigigabit transceivers RocketIO™ o RocketIO X empotrados que permiten comunicaciones de hasta 3.125 Gb/s por canal (RocketIO) o 6.25 Gb/s (RocketIO X).

Puede incluir hasta dos procesadores RISC PowerPC405. Las principales características del procesador empotrado son velocidad superior a 300 MHz, bajo consumo (0.9 mW/MHz), camino de datos pipeline de 5 etapas, unidad

de multiplicación/división en HW, 32 registros de propósito general de 32bits, cachés de 16KB para datos e instrucciones, unidad de gestión de memoria (MMU) y soporte para desarrollo y pruebas.

La alimentación auxiliar es de 2.5V en lugar de 3.3 como en Virtex-II, y este cambio proporciona menor tamaño, mayor velocidad y menor consumo de potencia.

Las Virtex-II Pro no son compatibles ni en la configuración de pines ni en el mapa de bits con Virtex-II. Sin embargo, los diseños para Virtex-II se pueden compilar para Virtex-II Pro.

En cuanto a la memoria se mantiene la capacidad de la LUTRAM distribuida y se aumenta la de la BlockRAM hasta 8Mb.

Los IOBs tienen igual arquitectura y agrupamiento (en bloques alrededor de la FPGA). Se dispone de una terminación de línea On-chip para las entradas LVDS diferenciales.

Arquitectura de configuración de las Virtex-II/ Pro. Organización, estructura y direccionamiento.

En Virtex II la estructura de la memoria de configuración es igual que la de las Virtex. La principal diferencia está en el sistema de distribución de reloj.

La figura A1.14 muestra la arquitectura de configuración y el árbol de reloj de un dispositivo XC2VP7 de la familia Virtex-II Pro, donde se muestra a la derecha el contenido de un frame de configuración de una columna de CLBs. Los bits que corresponden al control del arbol de reloj se encuentran al principio y al final del frame, mientras que los correspondientes a los CLBs son contiguos y en posiciones fijas dentro del frame. Esto es importante ya que esta forma de distribución de reloj divide a la FPGA en cuadrantes que no tienen la altura de un frame.

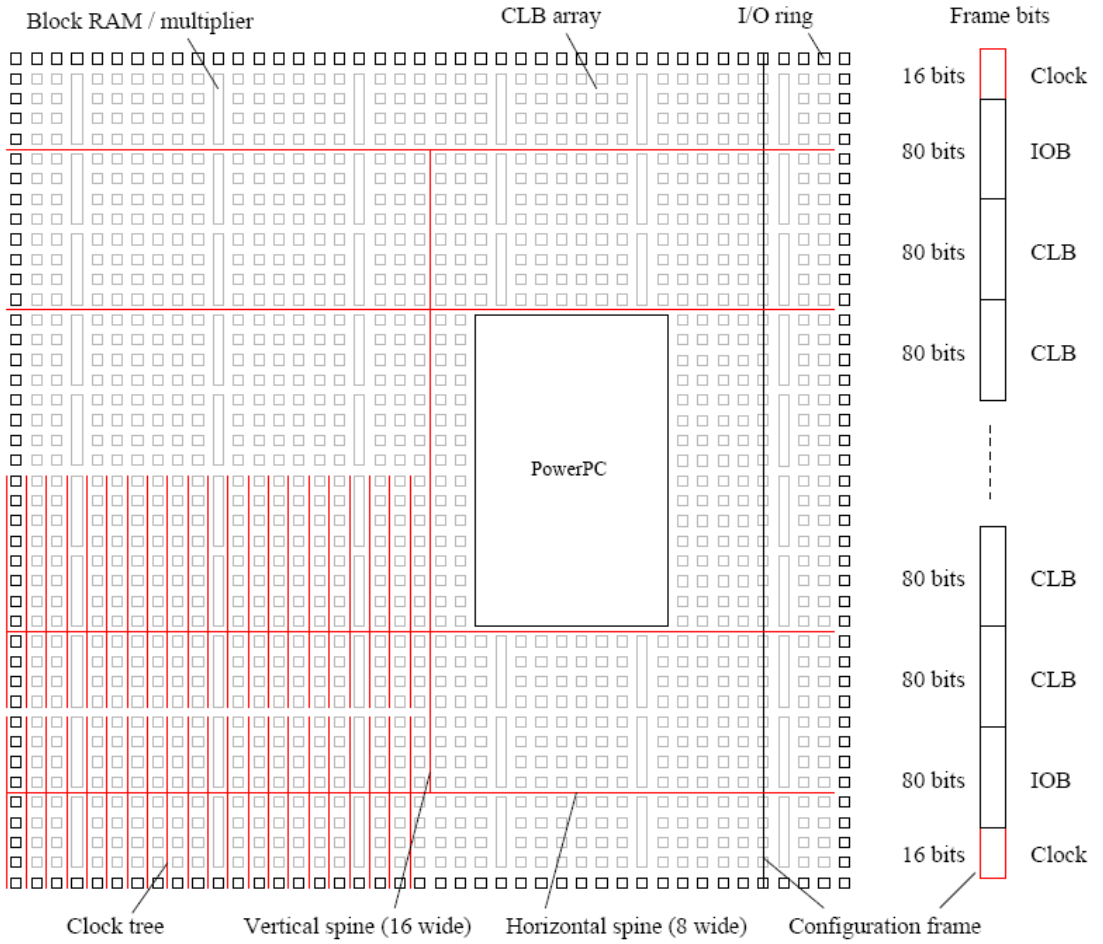


Figura A1.14. Arquitectura y frame de configuración para un dispositivo XC2VP7.

La reconfiguración dinámica se puede realizar a partir de la Virtex-II a través del puerto de configuración interna global ICAP (Internal Configuration Access Port), el JTAG o el SelectMAP. El puerto ICAP permite acceder a los datos de configuración del mismo modo que con SelectMAP, aunque utiliza puertos de E y S de datos separados, mientras que el SelectMap es bidireccional.

Como ya se planteó en el capítulo 1, para un modelo de arquitectura como el de las Virtex, Virtex-II y Virtex-II Pro, en el que el elemento básico reconfigurable dinámicamente es una columna, la asignación de una ubicación para la tarea tiene lugar mediante una técnica que gestiona el espacio libre únicamente en una dimensión (por columnas), tomando como unidad básica un tamaño de columna prefijado (variable para cada modelo

concreto de la familia). En este caso, las técnicas a utilizar se asemejarían bastante a las empleadas para gestión de memoria.

En el caso de configuración 1D cada tarea se debe modelar como una columna, sin embargo, la tarea únicamente utilizará un porcentaje de los recursos incluidos en dicha columna. El resto de recursos, que no se están utilizando de ninguna forma, no se pueden utilizar para implementar otras tareas, por lo que la fragmentación interna es mayor que con una gestión en 2D. En dispositivos de gran tamaño la fragmentación interna es mayor y puede haber problemas con la distribución de señales de alta velocidad. Las técnicas a emplear para realizar procesos de defragmentación son relativamente sencillas y se realiza una gestión menos eficiente de los recursos que en el caso 2D.

E. Virtex-4

Aparece a finales del 2004 y mejora mucho las posibilidades de programación respecto a familias anteriores, siendo una alternativa interesante frente a los ASIC. Los dispositivos se fabrican en un proceso de 90 nm y esta familia marca un cambio significativo en el layout respecto de las familias anteriores.

Comienza una tendencia hacia la personalización del producto como se predecía en las curvas de Makimoto, y aparecen tres subfamilias diferenciadas: LX, FX y SX, ofreciendo mayores posibilidades de elección para poder soportar todo tipo de aplicaciones complejas. Los bloques sintetizados dentro de la Virtex-4 incluyen el procesador PowerPC, Controladores de acceso a Ethernet tri-modo (MAC, de *Media Acces Controller*), transceivers serie hasta 6.5Gb/s, DSP dedicados (contienen un multiplicador de 18*18, un sumador y un acumulador de 48 bits) y circuiteria para gestión de reloj de alta velocidad.

CLBs. Los bloques básicos de la Virtex-4 son mejoras de los utilizados en las anteriores familias Virtex, por lo que los diseños previos se pueden actualizar y son compatibles. Continúa siendo el principal recurso lógico para implementar circuitos secuenciales o combinacionales. Cada CLB se conecta

a una matriz de conexión para acceder a la matriz de routing general, como se observa en la figura A1.15. Cada CLB contiene cuatro slices interconectados que se agrupan en parejas y cada pareja está organizada en una columna. En la figura se ve la disposición de los cuatro slices, donde SLICEM designa al par de slices situados en la columna de la izquierda, y SLICEL designa a la pareja de la columna derecha.

Cada pareja en una columna tiene una cadena de acarreo independiente, sin embargo, sólo la columna SLICEM tiene una cadena de desplazamiento común.

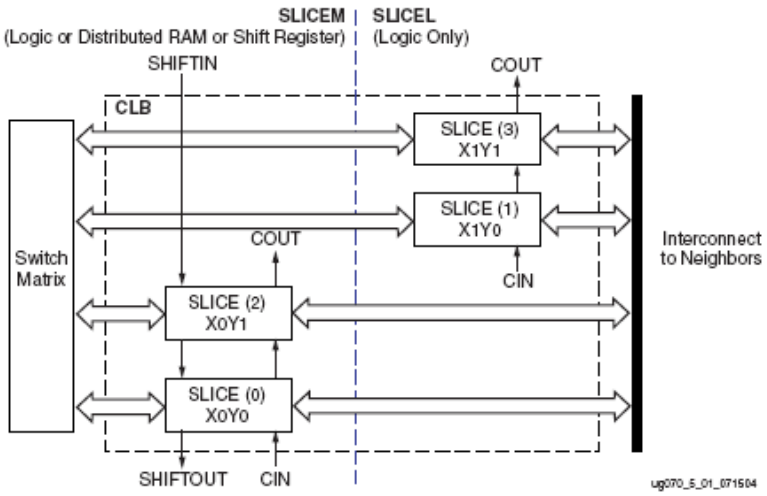


Figura A1.15 . Detalle de CLB.

Los elementos comunes a las dos parejas de slices (SLICEM y SLICEL) son similares a familias anteriores. El SLICEM soporta además dos funciones adicionales: almacenamiento de datos usando RAM distribuida y desplazamiento de datos usando registros de 16 bits.

Gestión de señales de reloj. La familia Virtex-4 mejora la distribución del reloj usando regiones de reloj en lugar de cuadrantes como en las anteriores familias. Cada región de reloj puede tener hasta 8 dominios de reloj global. Estos 8 relojes globales pueden ser conectados a cualquier combinación de los 32 buffers de reloj globales. Las reglas y restricciones que había en arquitecturas previas no se vuelven aplicar, en concreto, una región de reloj no esta limitada a cuatro cuadrantes sin tener en cuenta el tamaño del dispositivo. Ahora las dimensiones de una región de reloj están fijadas a una

altura de 16 CLBs (32 IOBs) y se expande en horizontal hasta la mitad del dispositivo como se puede ver en la figura A.16. Como las dimensiones de una región de reloj están fijadas, los dispositivos de mayor tamaño tendrán más regiones de reloj, y por tanto los dispositivos Virtex-4 pueden soportar muchos más dominios de reloj que las arquitecturas previas.

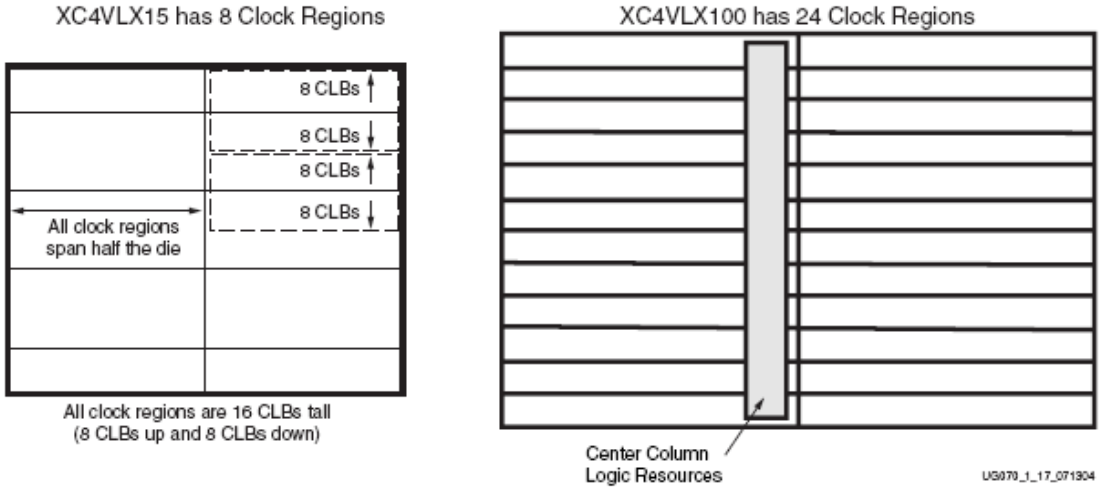


Figura A1.16 . Regiones de reloj en Virtex-4.

Los recursos que hay en la columna central (DCM, ÏOBs, etc) se sitúan en la región de reloj de la izquierda. Los DCMs, si se usan, utilizan los relojes globales de la región izquierda como líneas de realimentación y puede haber hasta 4 DCMs en una región especifica.

Arquitectura de configuración y reconfiguración dinámica.

La arquitectura de configuración de la Virtex-4 tiene diferencias importantes respecto a sus predecesoras. Primero, mientras la unidad mínima de configuración es todavía un frame, el frame ahora expande su altura a una región de reloj, en lugar de la altura total del dispositivo. Hay dos puertos ICAP en el dispositivo Virtex-4: el superior (*top*) e inferior (*bottom*) que no pueden estar activos al mismo tiempo, siendo el *top* el que queda activo por defecto después de la configuración inicial.

Los dispositivos Virtex-4 se dividen en dos mitades, superior e inferior. Las frames en la parte inferior son iguales a la superior con la excepción de la fila

de HCLK vertical que contiene los relojes globales y regionales. Todas la frames en la Virtex-4 tienen una longitud fija e igual de 1312 bits (41 palabras de 32 bits). Un frame configura un HCLK con 4 blocks RAM, 32 bloques de E/S o 4 DSPs. Una frame se referencia a través del Frame Address Register (FAR), y dicha dirección se divide en cinco campos: a) bit de selección de mitad superior/inferior, b) tipo de bloque (CLB/IO/CLK/blockRAM/DSP), c) dirección de fila de frames de 16 CLBs de altura, d) dirección de columna principal y e) menor (dentro de la columna principal).

En toda la familia Virtex, toda la información de configuración se envía a través del interfaz de configuración, que se puede usar para escribir o leer la configuración del dispositivo. El interfaz incluye hasta 14 registros de control de configuración y lógica para interpretar el mapa de bits.

La figura A1.17 muestra la arquitectura de configuración de la Virtex-4.

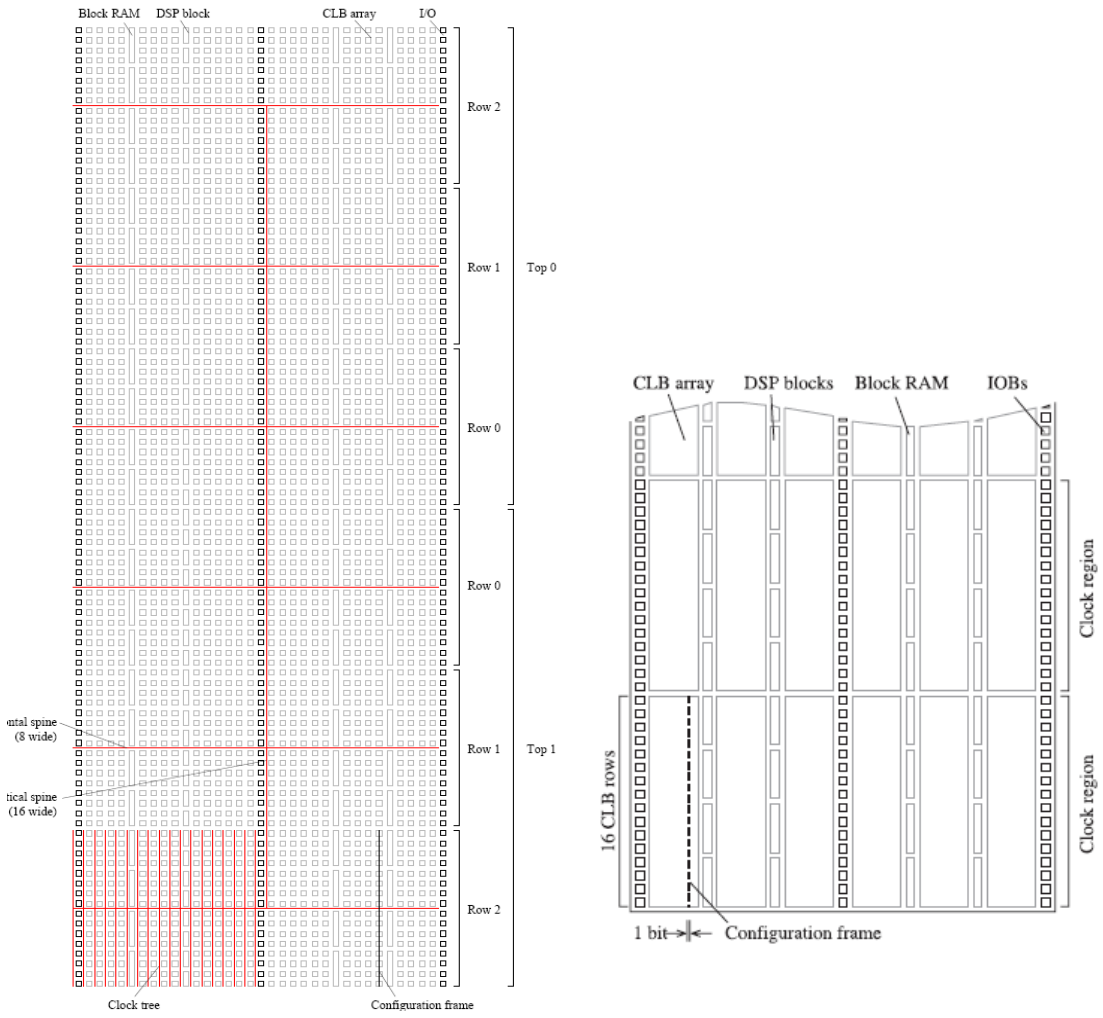


Figura A1.17. Arquitectura de configuración de la Virtex-4.

El cambio en la región que puede configurar un frame en la Virtex-4 (un frame ahora se expande a la altura de una región de reloj, en lugar de la altura total del dispositivo) es muy importante ya que permite y simplifica la gestión real en 2D, si hacemos coincidir una región de reloj con el tamaño mínimo de región reconfigurable.

Esta posibilidad de reconfiguración parcial dinámica se puede usar por ejemplo para especificar una variedad de condiciones estáticas en bloques funcionales como DCMs y RocketIO Multigigabit Transceivers (MGTs). Cada bloque funcional que puede requerir reconfiguración dinámica tiene una memoria de configuración con los siguientes atributos: es directamente accesible desde el dispositivo, cada bit de la memoria se inicializa con el valor correspondiente del bit de configuración en el mapa de bits, y los bits de la memoria se pueden posteriormente cambiar a través de ICAP. Cada bit de la memoria controla la lógica del bloque funcional, de modo que el contenido de esta memoria determina la configuración del bloque funcional.

F. VIRTEX-5

Esta familia es una evolución de la Virtex-4 y mantiene su arquitectura y características generales. Se fabrica en tecnología de proceso de 65nm, proporciona mayor densidad y rendimiento con menor consumo y tiene V_{CC} a 1V en lugar de 1.2V. Aparece una nueva familia LXT para aplicaciones lógicas con conectividad serie avanzada. Si comparamos la Virtex-5 con la Virtex-4 las mejoras son las siguientes:

Las LUTs tienen 6 entradas independientes frente a las 4 de Virtex-4 y por tanto se dispone de mayor densidad. Las LUT son 1 o 2 grados más rápidos que en Virtex-4 y además pueden configurarse como funciones de 6 entradas o como 2 funciones de 5 entradas.

Los SRL se pueden encadenar hasta conseguir 128 bits en un CLB, frente a los 84 de la anterior Virtex-4, por tanto se dispone de pipelines de mayor profundidad.

Cada Slice DSP48E tiene un multiplicador de 25*18, un sumador y un acumulador. Existe posibilidad de encadenamiento de carry por columnas de

slices. Cada Slice puede tener uno o ningún Slice M. Cada LUT de un SliceM puede operar como memoria distribuida de 32 bits y cada SliceM puede configurarse como un registro de desplazamiento de 32 bits.

Los DSPs se aumentan en número y velocidad y pasan de 512 slices a 500MHz a 640 slices a 550MHz. Además se reduce el consumo de 2.3 mW/100 MHz hasta 1.38 mW/100 MHz.

Cada bloque de BlockRAM aumenta de tamaño de 18 a 36 Kbits (aunque pueden utilizarse como 2 bloques independientes de 18Kbits) y de velocidad de 500 a 550 MHz. La BlockRAM y la FIFO disponen ahora de un código de detección y corrección de errores (ECC). Los bloques aumentan de tamaño de 18 a 36 Kbits y de velocidad de 500 a 550 MHz.

La interconexión en V-5 permite routing diagonal, en lugar del segmentado de la V-4, proporcionando conexiones más rápidas.

Cada CMT (*Clock Manager Tile*) contiene 2 DCMs y un PLL. En la gestión de reloj de eleva la velocidad de 500 a 550 MHz, y se usan también PLLs que tienen menor jitter.

En la E/S se dispone de hasta 1.25Gbs diferencial frente a 1Gbs y tiene mayor capacidad de E/S, con un número de bancos de E/S de 33, casi todos con 40 E/Ss, 20 CLBs y una región de reloj (salvo los bancos de la columna central con 20 E/S).

Aparece un nuevo monitor del sistema, no disponible en la V-4, que gestiona térmicamente el chip y monitoriza las tensiones.

Arquitectura de configuración y reconfiguración dinámica.

En la configuración del dispositivo aparecen nuevos interfaces serie y paralelo. El serie se denomina SPI (*Master Serial Peripheral Interface*) Flash y el paralelo está disponible en dos modos BPI-Up (*Master Byte Peripheral Interface Up Flash*) y BPI-Down (*Master Byte Peripheral Interface Up Flash*). En el modo SPI se puede configurar a través del estandard de industria SPI para memorias Flash PROM (aunque SPI es un interfaz de cuatro hilos). El nuevo interfaz paralelo BPI permite configurar los dispositivos Virtex-5 a través del estandard paralelo para memorias NOR Flash PROM. La FPGA

puede direccionar hasta 26 lineas de direcciones y se soportan anchos de bus de 8 y 16 bits, que se autodetectan. Dicho interfaz permite dos modalidades BPI-Up (la direccion comienza en 0 y va incrementando) y BPI-Down (comienza en la dirección máxima y va decrementando).

Las frames en la Virtex-5 mantienen la longitud fija e igual de 1312 bits (41 palabras de 32 bits). Un frame configura una región de reloj HCLK que dependiendo del tipo puede contener 20 CLBs, 40 bloques de E/S, 4 DSPs o 4 blocks RAM.

La figura A1.18 muestra en detalle el orden y el contenido de un frame que configura los 20 CLBs de una región de reloj de las 8 que contiene un dispositivo XC5VLX30.

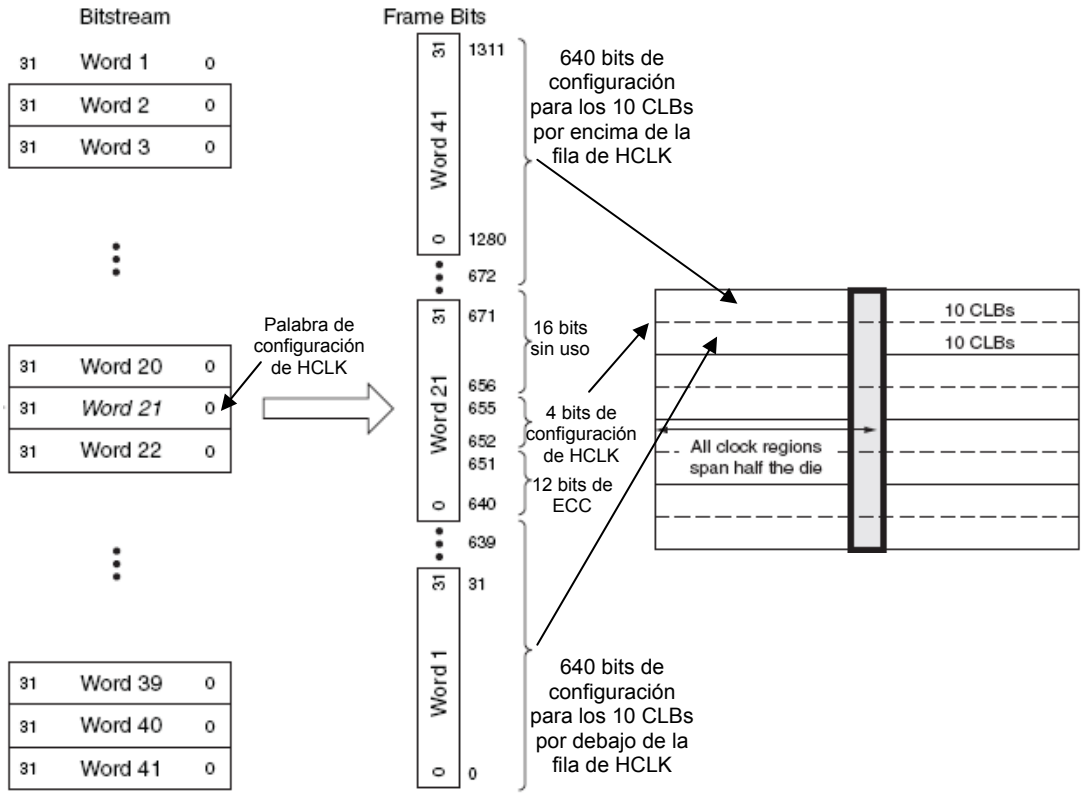


Figura A1.18. Arquitectura de configuración de la Virtex-5.

En la tabla A1.1 se resumen de las principales características de las familias Virtex.

Tabla A1.1. Principales características de las familias Virtex.

	VIRTEX	VIRTEX-II/PRO	VIRTEX-4	VIRTEX-5
Tecnología	220nm	130nm	90nm	65nm
Tamaño	1 M puertas 27K celdas	8 M puertas 100 K celdas	200K celdas	331 Kceldas
Max. LUTRAM	390Kb	1465/ 1378Kb	1392Kb	3420Kb
BlockRAM	131Kb	3.024/ 7.992Kb	9.936Kb	11.664 Kb
Multiplicadores y DSP	No	168/444 multiplicadores 18*18	512 DSP48	640 DSP48E
Modos de configuración	- Master Serie - Slave serie - SelectMAP - Boundary-scan	- Master Serie - Slave serie - M SelectMAP - M SelectMAP - Boundary-scan	- Master Serie - Slave serie - M SelectMAP - M SelectMAP - Boundary-scan	- Master Serie - Slave serie - M SelectMAP - M SelectMAP - Boundary-scan - SPI - BPI-Up - BPI-Down
Puertos de configuración	SelectMap 8bit	JTAG, ICAP y SelectMap 8bit	JTAG, ICAP y SelectMap 8 y 32bit	JTAG, ICAP y SelectMap 8 y 32bit
Tamaño máximo del bitstream	6Mb	29,1/ 33,5 Mb	51,4Mb	79,7 Mb

A1.2 Comunicación de tareas

La comunicación de tareas o módulos reconfigurables, en las familias Virtex y Virtex-II/Pro se puede realizar a través del bus macro triestado descrito en [Xili04b]. Los bus macros se utilizan como rutas de datos fijas entre dos módulos reconfigurables o entre un módulo reconfigurable y uno fijo. Cualquier señal de comunicación entre módulos debe pasar a través de un

bus macro. Tienen un rutado y una ubicación siempre fijas que no deben cambiar cuando se reconfigura un módulo o tarea.

En la figura A1.19 aparece un esquema detallado de la interfaz de un bus macro basado en buffers triestado (TBUF) entre dos regiones reconfigurables B y C. Para enviar una señal desde la izquierda (región B) hacia la derecha (región C), por ejemplo para la entrada LT[0] se pone a nivel activo y la señal LI[0] se transmite a la linea RO[0] que conecta las dos caras de los módulos. En la dirección contraria es posible enviar una señal si se habilitan los buffers de la región C y se deshabilitan los de la región B.

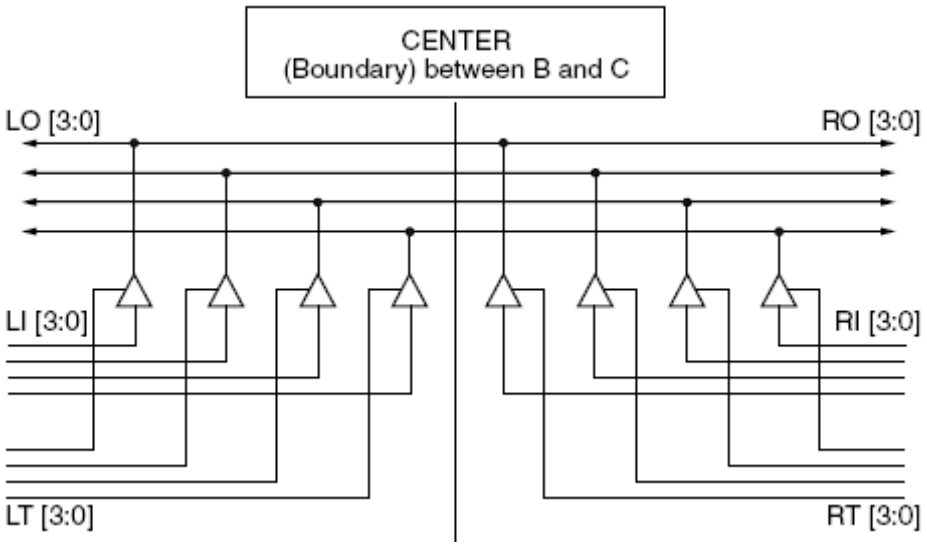


Figura A1.19. Bus macros de familias Virtex.

Cuando se realiza la comunicación entre tareas en dispositivos de las familias Virtex y Virtex-II/Pro se deben tener en cuenta las siguientes restricciones:

La anchura de cada tarea va desde un mínimo de 4 slices a un máximo de la anchura de todo el dispositivo, en incrementos de 4 slices.

Los límites definidos de región reconfigurable no pueden cambiar dinámicamente y la posición que puede ocupar una tarea dada es siempre la misma.

Cada bus macro proporciona 4 bits de comunicación entre módulos, de modo que si un módulo A comunica con otro módulo B a través de 32 bits, se necesitan 8 bus macros.

En la figura A1.20 se muestra una comunicación entre una tarea A y otra B vecinas a través de un bus macro. Los bus macros también tienen que utilizarse cuando la señal pasa a través de un módulo conectando los módulos situados en sus extremos, como en la figura A1.20 que hay una comunicación entre la tarea A y C a través de la tarea B. En los dos casos se utilizan recursos de interconexión que no pueden cambiar cuando el módulo B se reconfigura. Esto supone que para realizar el diseño del módulo B se debe conocer los requisitos de comunicaciones de los módulos que comunican a través de el. Como se observa en la figura, estos "puentes de comunicación fijos" están prerutados y utilizan los buffer triestado asociados a cada CLB y las líneas largas segmentables. El número de canales de comunicación disponibles está limitado por el número de recursos de interconexiones horizontales largas disponibles en cada fila (*tile*) de CLBs. Este método de comunicación está descrito en profundidad en la XAPP290 de Xilinx [Xili04b].

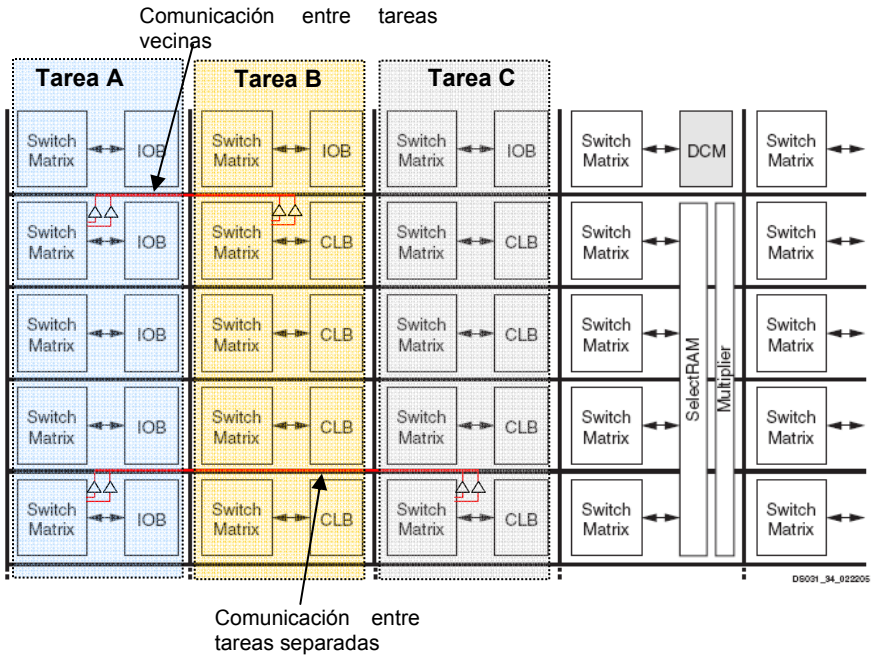


Figura A1.20. Bus macros de familias Virtex.

H. Kalte en [KKK04] también describe otro sistema de reconfiguración en el que la comunicación también se basa en el uso de buffers y lineas triestado, pero en el que las tareas pueden tener una anchura arbitraria. Dicho sistema se mostró en detalle en el capítulo 2.

En las familias posteriores como las Virtex 4 y 5 la comunicación con bus macro basados en TBUF desaparece, ya que no tienen líneas ni buffers tri-state. Para estas familias el bus macro se implementa con slices y se denomina Slice-based macro. Este tipo de comunicación también se puede implementar en la familia Virtex-II.

En la figura A1.21 se muestra el esquema de interconexion simplificado entre dos tareas A y B donde aparecen dos señales de comunicación (una de entrada y otra de salida) entre dos posibles tareas A y B. En la dirección de izquierda a derecha, la señal generada por la tarea B entra por la entrada *G4* y sale por la salida *Y* del slice de la columna *X* y entra por la la entrada *G4* y sale por la salida *Y* del slice de la columna *X+1*, mientras en la dirección contraria entra por F4 del slice de la columna *X+1* y sale por la salida X del slice de la columna *X* (tarea B). En ambos slices las LUTs están preasignadas para que la entrada G4 salga por Y y para que F4 salga por X, pero es posible cualquier otra combinación. En este esquema se ha obviado la representación de las matrices de interconexión asociadas a los CLBs de las columnas X y X+1. Para cada pareja de CLBs (en la misma fila o columna) se puede implementar cualquier combinación de entradas y salidasde hasta 8 bits.

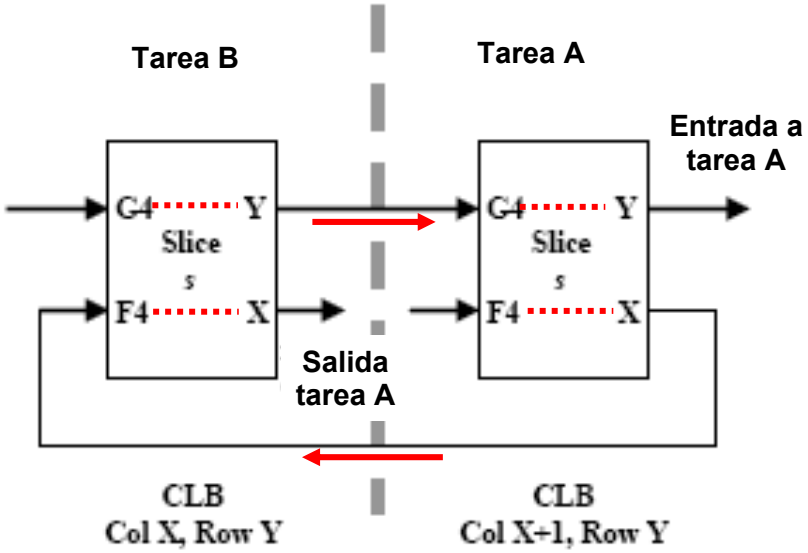
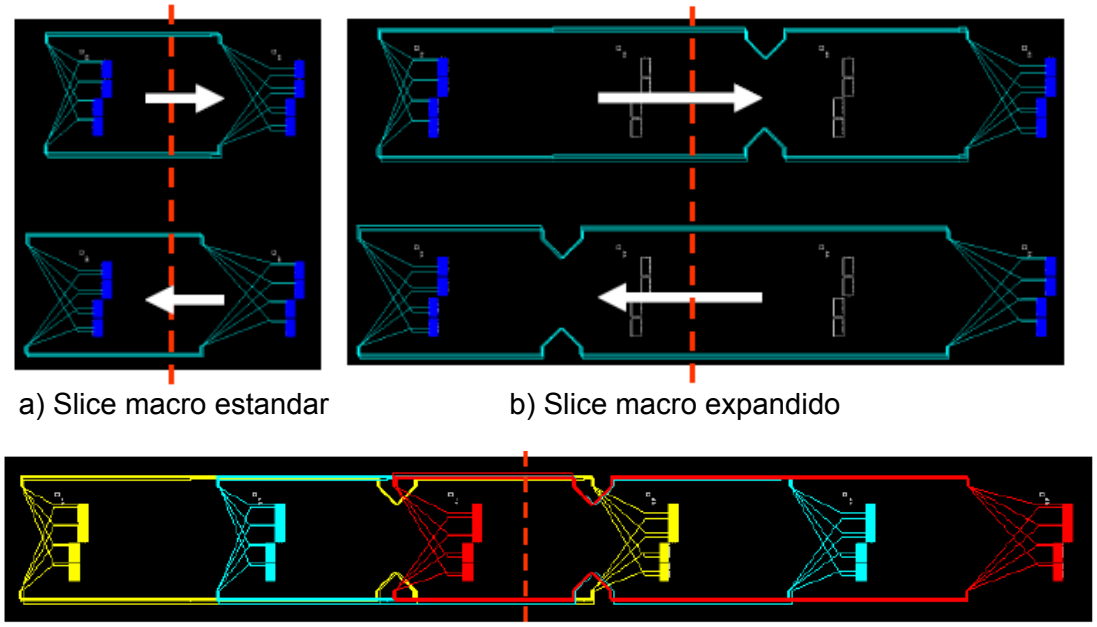


Figura A1.21. Slice based macros con una entrada y una salida.

La figura A1.22 muestra diferentes tipos de bus macros basados en slice donde aparece la matriz de interconexión. En (a) aparece un bus macro estandar entre CLBs adyacentes donde en la pareja superior se dispone de 8 señales de comunicación de izquierda a derecha y en la pareja inferior de otras 8 señales en la dirección opuesta. En (b) aparece un ejemplo similiar en el que los CLBs no están en la frontera de la region reconfigurable, sino en el interior. En (c) aparecen tres parejas de CLBs que comunican de manera intercalada, proporcionando un total de 24 señales de comunicación (8 por cada pareja de CLBs). Estos esquemas de conexión se pueden implementar también entre CLBs dispuestos en la misma columna. Los bus macros basados en Slice proporcionan por tanto mayor densidad de señales y con mayor variedad de ubicaciones de los CLBs ya que no tiene la restricción de estar en la frontera.



a) Slice macro estandar b) Slice macro expandido

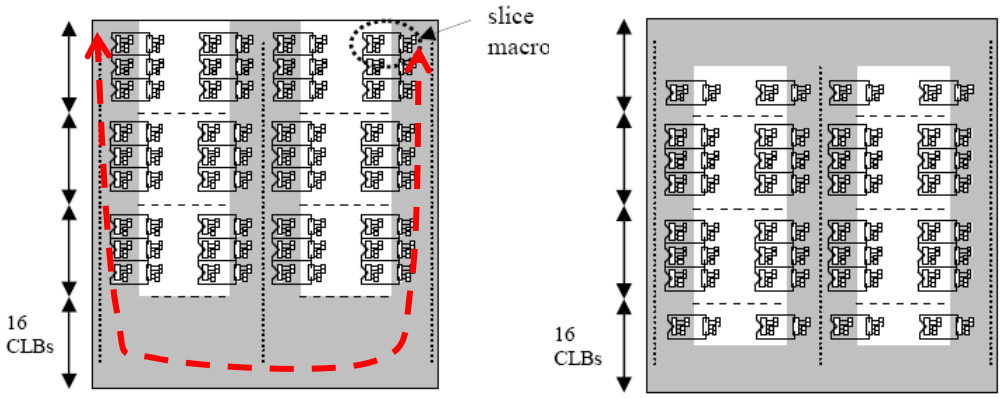
c) Slice macro intercalado

Figura A1.22. Tipos de Slice based macros (a) estándar (b) expandido e (c) intercalados.

Para las familias Virtex 4 y 5 se han propuesto diversas soluciones como la propuesta por P. Sedcole [SBBA06] donde se presenta una aplicación práctica denominada Sonic-On-a-Chip (arquitectura reconfigurable para procesamiento de imágenes de video) donde en lugar de los buffers triestate (familia Virtex) se utiliza una estructura de bus con lógica-OR implementada sobre una Virtex-4 XC4VLX25.

En el trabajo de Koh y Diessel [KoDi06] se muestra una nueva metodología para el desarrollo de una infraestructura de comunicaciones entre módulos que utiliza bus macros basados en slice, implementados en una Virtex-4. En la figura A1.23 se muestran las dos opciones del layout de la infraestructura de comunicaciones. La zona gris, donde reside la infraestructura, envuelve todos los IOB externos. De los dos tipos de layout presentados, el de forma de tridente en (a) tiene la ventaja de ser más simple porque todos los módulos reconfigurables son iguales, pero puede presentar mayores retardos en el camino crítico, mostrado con línea roja punteada. El otro layout,

mostrado en (b), tiene menores retardos a costa de mayor complejidad en la reubicación, ya que hay tres tipos de áreas de asignación: las dos de la parte superior, las cuatro de la zona intermedia y las dos de la zona inferior, sólo se podrán reubicar módulos en áreas del mismo tipo.



a) Layout Tridente b) Layout Doble

Figura A1.23. Alternativas para implementación de infraestructura de comunicaciones.

En la figura A1.24 se muestra en detalle la infraestructura de la parte izquierda de un módulo. En la figura se detalla cómo se implementa una entrada y una salida dentro de un slice.

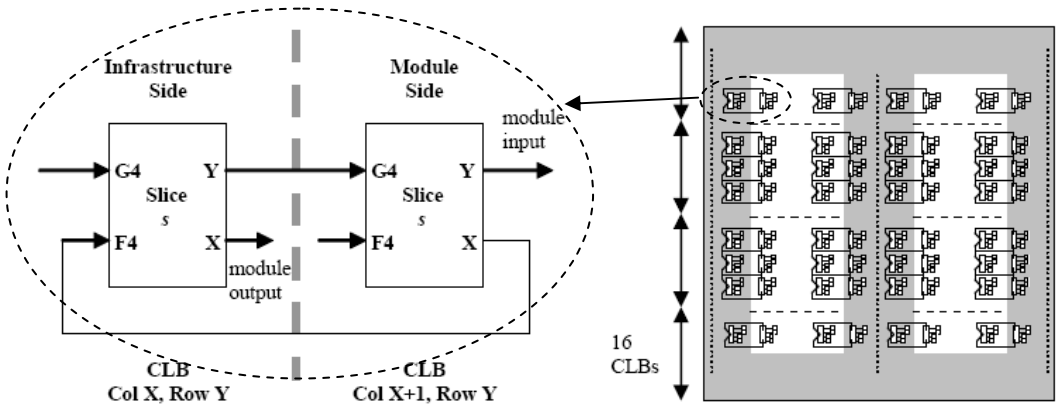


Figura A1.24. Detalle de infraestructura de comunicaciones

Además cada área de módulo es capaz de conectar un gran número de bits a la infraestructura de comunicaciones. Hay un máximo posible de 8x2x16=256 bits de comunicación para cada módulo cuando se usan todos los CLBs de sus columnas más a la izquierda y derecha para implementar los slice based macros.

Como comentario general se podría añadir la falta de soporte oficial para poder realizar la reconfiguración dinámica en estos dispositivos, ya que hasta el momento no hay herramientas comerciales ni ninguna nota de aplicación proporcionada por Xilinx.

Apéndice 2:

Perspectivas de evolución de las FPGAs

Algunos comités, como el International Technology Roadmap for Semiconductors ITRS [ITRS05] identifican, entre otras, las capacidades o funcionalidades técnicas que debe desarrollar la industria del Semiconductor para continuar las tendencias de crecimiento. El ITRS elabora hojas de ruta (*roadmaps*) en las que se identifican las necesidades tecnológicas principales para guiar las investigaciones compartidas entre la Industria, la Universidad y los Laboratorios Nacionales. Este Comité resalta dos puntos clave para el progreso:

1. Se detecta una previsión de los factores de escala más agresivos de los proyectados. Estas previsiones de factores de escala mantienen el progreso de la Ley de Moore (capacidad de la industria para doblar el

número de transistores de un chip cada dos años). Por ejemplo, en 2004 se realizaron dispositivos de memoria RAM con tecnología de 90 nanometros cuando lo proyectado era 100 nanómetros en 2005, según lo publicado en la hoja de ruta justo cuatro años antes. De un modo similar, la tecnología de fabricación de los transistores de un microprocesador, una dimensión crítica que afecta a la velocidad del procesador, es de 25 nanometros en 2008, seis años antes de lo esperado en la versión de la hoja de ruta de 1999.

2. En unos 15 años está previsto alcanzar los límites fundamentales de los materiales usados en los procesos CMOS planares (proceso que ha sido la base de la industria del semiconductor durante los últimos 30 años). Con la introducción de nuevos materiales en la estructura básica CMOS e inventando nuevas estructuras de CMOS, las mejoras en los procesos CMOS pueden continuar durante los próximos 10 o 15 años. Para poder continuar y conseguir avances en la tecnología, será necesario investigar nuevos dispositivos que puedan proporcionar una alternativa de mejor coste efectivo que la CMOS planar.

Si tomamos juntas estas dos recomendaciones del ITRS se llega a la conclusión de que si se continúan acelerando los progresos técnicos en las mismas proporciones, se alcanzarán los límites de los procesos actuales muy pronto. Además no hay muchas soluciones de fabricación que sean una alternativa real y viable a los procesos actuales, al menos a corto plazo.

Para superar las limitaciones de las tecnologías de fabricación y arquitecturas actuales de los dispositivos reconfigurables, las principales alternativas o tendencias tecnológicas se pueden resumir en:

- **Nanotecnología y electrónica molecular**. La electrónica molecular se plantea como una alternativa a medio plazo a los procesos actuales de fabricación no sólo para reducir la escala del sistema, sino porque pueden aportar grandes beneficios debido a sus características.

- **FPGAs 3D optoelectrónicas.** Debido a los grandes retardos entre las conexiones dentro del chip, y que este recurso es el que más espacio ocupa dentro de la FPGA, se propone la utilización de enlaces ópticos, con mayor ancho de banda y menores retardos, en sustitución de las interconexiones actuales basadas en hilos conductores. De esta manera se reduce el consumo, espacio, y latencia de las cargas de configuración a la vez que se mejoran las prestaciones.
- **FPGAs 3D**. Hay alternativas arquitectónicas como las FPGAs tridimensionales, formadas por varios niveles de chips interconectados verticalmente. Se consigue reducir el espacio y la distancia del rutado en sistemas complejos y grandes.
- **FPGAs con memoria de configuración no volátil**. Las FPGAs basadas en memoria no-volátil almacenan los datos de configuración dentro del chip eliminando la necesidad de memoria externa y de los largos tiempos de configuración asociados a las FPGAs basadas en SRAM.

A2.1 Nanotecnología y electrónica molecular

Se pueden construir dispositivos basados en moléculas individuales o nanotecnología capaces de realizar funciones idénticas o análogas a las realizadas por dispositivos microelectrónicos tales como diodos, transistores, conductores y resistencias. Entre los dispositivos que se han experimentado en el campo de la microelectrónica se encuentran los nanotubos y nanohilos.

Los **nanotubos** de carbono (NTs) son una forma alotrópica del carbono (el elemento existe en diferentes estructuras moleculares), como el diamante, el grafito o los fulerenos. Su estructura puede cosiderarse procedente de una lámina molecular de anchura atómica de grafito enrollada sobre sí misma. Dependiendo del grado de enrollamiento, y la manera como se conforma la lámina original, el resultado puede llevar a nanotubos de distinto diámetro y geometría interna. La geometría y la mayoría de las propiedades de los nanotubos de carbono dependen de su diametro y de su angulo chiral,

también llamado helicidad. Sus dimensiones pueden alcanzar 1 nm de ancho, que puede tener varios milímetros de largo. Puesto que es una molécula simple, los NTs son extremadamente fuertes y flexibles [CoAA01]. Los nanotubos se pueden comportar como un metal conductor o como un semiconductor, dependiendo de su geometría atómica.

Atendiendo al número de capas se pueden clasificar en:

- Nanotubos de capa múltiple (MWNT): son aquellos formados por capas concéntricas de forma ciclíndrica, las cuales están separadas aproximadamente una distancia similar a la distancia interplanar del grafito.
- Nanotubos de capa única (SWNT): Son los que se pueden describir como una capa bidimensional de grafito enrollada formando un cilindro de décimas de micrones de longitud y radio del orden de los nanómetros. Se conocen derivados en los que el tubo está cerrado por media esfera de fulereno, y otros que no están cerrados.

Atendiendo a una clasificación genérica:

- Nanotubos chiral; no tiene simetría de reflexión y son no isomorficos.
- Nanotubos no-chiral; (zigzag y armchair) poseen simetría de reflexión y son isomórficos.

La figura A2.1 muestra los tres tipos de construcción de nanotubos de carbón.

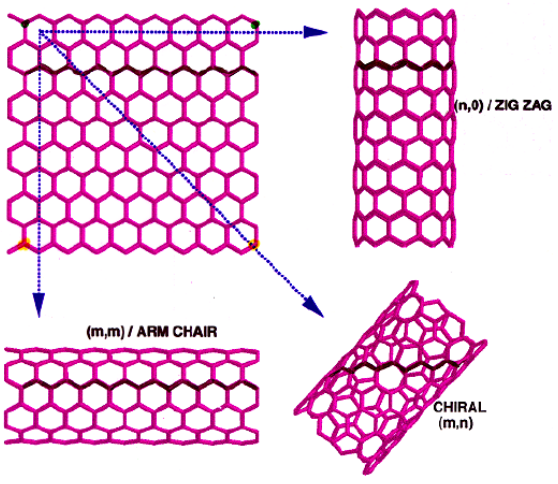


Figura A2.1. Tipos de nanotubos de carbón.

La producción del tipo MWNT es más compleja y costosa, por lo que los SWNT son más adecuados para realizar componentes microelectrónicos debido además a sus propiedades eléctricas. El tipo mas básico seria un conductor, pero se pueden realizar también otros tipos de dispositivos más complejos como FETs.

La figura A2.2 muestra la estrutura e imagenes de un FET de nanotubo (CNT-FET), donde en (A) aparece la estructura de un CNT-FET sobre un substrato de silicio, con los electrodos metálicos de fuente y drenador situados en ambos extremos de la región del CNT. Además se muestran los electrodos de puerta superior (alrededor del CNT) y de puerta inferior (por debajo del substrato de silicio) que controlan la corriente que pasa a través del CNT. En (B) se muestra la imagen microcopica de un prototipo de CNT-FET, donde se puede apreciar cada uno de los disitinctos componentes salvo el propio tubo CNT porque está oculto por el electrodo de puerta. En (C) aparece la imagen escaneada donde aparece el CNT visible entre los electrodos.

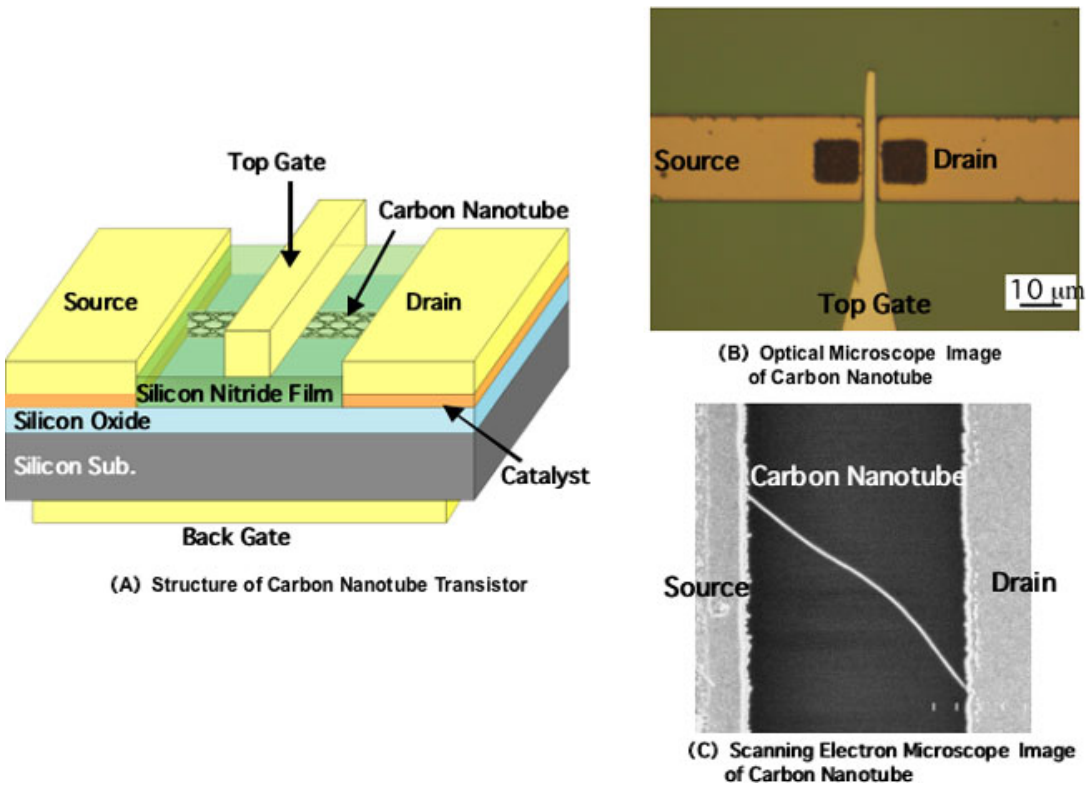


Figura A2.2. Estructura de CNT-FET.

Además se los CNTs se pueden utilizar para formar arrays programables de tipo *crossbar* como se muestra en la figura A2.3. Esta tecnología es la más reciente y con el tiempo se espera que sea más manejable y predecible.

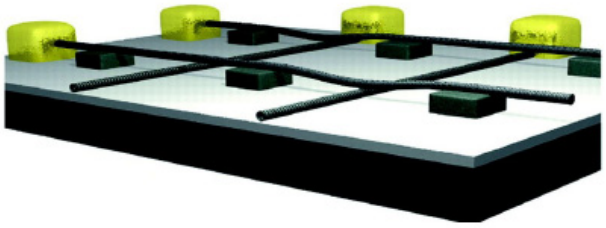


Figura A2.3. Crossbar realizado con nanotubos.

Los **nanohilos** (NH) pueden crecer en dimensiones controladas en la escala de nanometros para determinar su diámetro, pudiéndose conseguir nanohilos con 3nm de diámetro. Mediante un proceso de recrecimiento y controlando el perfil de dopaje se pueden formar aislantes (controlando el espacio entre conductores y entre hilos de puerta o de control). También se pueden formar semiconductores de tipo N y P, usando nanohilos de silicio con un dopado controlado de boro y fósforo [CDHL00] e incluso puertas basadas en diodos NH.

El interés se ha concentrado fundamentalmente en la producción de nanohilos de materiales conductores, semiconductores y óxidos de semiconductores.

Su aplicación para la fabricación de dispositivos reconfigurables se orienta a disminuir las interconexiones de los dispositivos tipo FPGA por el alto porcentaje de superficie que se emplea en la FPGA para interconexionado y rutado y que suele representar entre el 70 y 80% de la superficie del dispositivo. Un porcentaje aproximado se disipa en potencia en la parte de rutado del dispositivo. Por tanto, la electrónica molecular tiene la ventaja de no sólo reducir la escala del sistema, sino también, debido a su característica de poder almacenar su propio estado, las FPGAs basadas en esta tecnología no necesitarían memoria de configuración adicional.

Los tipos de switches que se pueden fabricar son:

- moleculares: cuando se aplica una tensión a la molécula se añade o se resta un electrón y se cambia su conductividad. Un ejemplo se encuentra en [TVHH02]. Estas moléculas se autoposicionan en los puntos de cruce de los nanohilos, y se pueden cambiar entre los estados de ON y OFF aplicando una tensión de polarización, por lo que su programación se realiza como en las FPGAs antifuse, pero con la ventaja de ser reprogramables y evitando la necesidad de tener memoria SRAM para almacenar los datos de configuración.
- mecánicos: utilizando nanotubos de carbono dispuestos en barra de cruce, de modo que cuando se aplica una tensión entre dos NTs se atraen, quedándose en contacto, y cuando se aplica una tensión de polaridad inversa se vuelven a apartar. La figura A2.4 muestra una conexión basado en el empleo de NTs suspendidos.

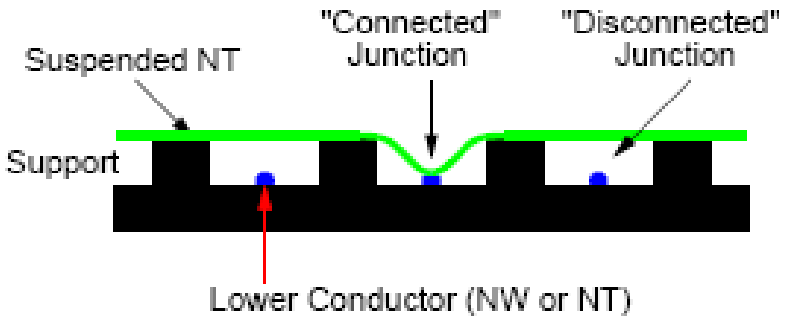


Figura A2.4. Conexión realizada con NTs suspendidos.

Procesos de fabricación, técnicas de montaje y estructuras.

Una de las principales diferencias entre la electrónica molecular y la tradicional VLSI reside en su tecnología de fabricación: la molecular está basada en la fabricación de *bottom-up* (primero se fabrican los dispositivos individuales e hilos y después se montan en sistemas), en oposición a los tradicionales *top-down* usados en la fabricación de los circuitos actuales.

La reducción en el tamaño de los conductores estará acompañada de varios problemas en la fabricación, como la limitación de sólo poder generar *layouts*

regulares (es muy difícil crear *layouts* arbitrarios), nuevos procedimientos para alinear los nanohilos y un mayor nivel de defectos en la fabricación a medida que va decreciendo el tamaño. Sin embargo, para poder usar de manera eficiente esta tecnología, es necesario mejorar las técnicas de montaje y alineación.

Muchos investigadores como Goldstein [Gold05] [GoBu01], DeHon [DeWi04] o Gayasen [GaVI05] han propuesto distintas arquitecturas mediante el uso de este tipo de dispositivos como alternativa a los métodos actuales de fabricación del rutado de los dispositivos reconfigurables.

Goldstein y Budiu describen una arquitectura denominada nanofabrica en [GoBu01], que es una malla bidimensional de nanobloques. El layout de una nanofabrica es similar a una FPGA de grano fino tipo isla. Cada *cluster* contiene 128 nanobloques localmente conectados a través de bloques de interruptores (switch-blocks). Los nanobloques están realizados con nanohilos situados en 2 capas consecutivas que forman 90 grados y cuyos puntos de cruce se pueden programar.

DeHon muestra en [DeWi04] cómo construir una arquitectura universal basada en FETs de nanohilos de silicio. Los nanohilos de silicio se organizan en nano-arrays que realizan funciones lógicas, y estos nano-arrays pueden contener entre 100 y 1000 NHs de alto y ancho, dependiendo de las necesidades de amortización del coste de programación y las proporciones de defectos que puedan contener. Cada nanohilo pasa por múltiples nano-arrays para realizar dos funciones: proporcionar interconexión entre arrays y realizar lógica. La arquitectura resultante se puede ver como un array de bloques PLA similar a una CPLD.

Las dos arquitecturas anteriores tienen muchas características comunes:

- La unidad atómica de la lógica está basada en mallas bidimensionales.
- Las conexiones entre las mallas se realizan a través de hilos en nanoescala (NH). Los circuitos se crean por reconfiguración post-fabricación.

Además, los componentes de los circuitos se forman con dispositivos que realizan operaciones lógicas y otros dispositivos que proporcionan ganancia y aislamiento de la E/S. Los componentes moleculares automontados se soportan con estructuras creadas a escala litográfica (p.e. CMOS). Finalmente los hilos de las filas (y de las columnas) son equivalentes y los defectos se pueden evitar intercambiando la funcionalidad entre los hilos de una fila (o de una columna).

Dentro de las aplicaciones prácticas se debe citar la desarrollada por investigadores de los Laboratorios HP que han usado una técnica con la que se podría incrementar la densidad de los dispositivos programables tipo FPGA (o reducir su tamaño y consumo). Esta técnica se combina con la tecnología CMOS convencional y se depositan en la parte superior elementos de interconexión basados en NH, usando una arquitectura denominada *"field programmable nanowire interconnect (FPNI)"*. Puesto que las FPGAs convencionales utilizan entre el 80 a 90 por ciento de su CMOS para rutar señales, los circuitos FPNI son mucho más eficientes; la densidad de transistores en la actualidad usados para realizar la lógica es mucho mayor y la cantidad de potencia eléctrica necesaria para las señales decrece. Actualmente se está trabajando en un prototipo usando hilos crossbar de ancho 15 nanometros combinado con CMOS de 45 nanometros, que sería viable tecnológicamente para 2010. Esto seria equivalente a un avance de tres generaciones en el ITRS si no se tuviera esta tecnología.

Para el futuro se prevén modelos basados en tecnología con hilos de 4,5 nanometros de ancho, que estarían disponibles para 2020. La arquitectura de 4,5 nanometros combinada con CMOS de 45 nm podría estar en una FPGA hibrida de un tamaño del 4% del tamaño de una FPGA fabricada solamente con CMOS de 45nm (o diez veces la densidad). El ahorro en tamaño y potencia a estos niveles serán el objetivo en los próximos diez años.

A2.2 FPGAs 3D optoelectrónicas

Aunque los enlaces ópticos han sido utilizados para interconexiones de larga distancia y gran ancho de banda (LAN y WAN), siguen siendo útiles en corta distancia y conexiones masivamente paralelas, como en sistemas multiprocesador dedicados a ejecutar algoritmos con un alto grado de paralelismo. Sin embargo, para interconexiones entre procesador y memoria o entre los niveles jerárquicos de un sistema de memoria, es muy crítico debido a la latencia del enlace óptico. Cuando se utilizan los enlaces ópticos en FPGAs a nivel lógico para interconectar puertas proporcionan muchas ventajas.

Por una parte, como son dispositivos flexibles y totalmente programables, las velocidades son menores, debido a las capacidades y resistencia de los switches programables.

Debido a los grandes retardos entre las conexiones dentro del chip, e incluso entre chips, los periodos de reloj de implementaciones en una FPGA pueden llegar hasta decenas de nanosegundo. En este rango de valores de retardo, los enlaces ópticos de algunos nanosegundos de latencia pueden ser una alternativa válida.

Por tanto, se pueden construir arquitecturas tridimensionales que tengan muchas ventajas sobre las actuales de dos dimensiones, aunque su rendimiento dependerá de la latencia de los enlaces ópticos utilizados.

Dambre explora en [DaVV99] posibles arquitecturas de FPGAs tridimensionales y se estudia el impacto que tiene la latencia de este tipo de interconexión en el rendimiento del dispositivo. En este trabajo se consideran arquitecturas multi-FPGA que consisten en varios planos interconectados opticamente, donde cada plano contiene un número igual de chips FPGA. Dentro de cada plano los chips se pueden interconectar a través de sus pads de E/S eléctricamente y entre planos los chips se conectan óptimamente para formar arrays o pilas, como se muestra en la figura A2.5.

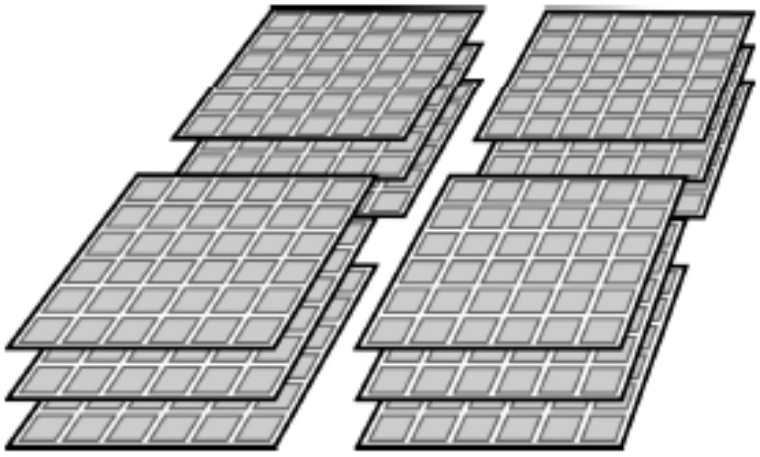


Figura A2.5. Multi-FPGA tridimensional con tres planos de 4 chips.

A2.3 FPGAs 3D

Aunque la idea de los circuitos 3D no es nueva, ciertos avances tecnológicos hacen que sea en la actualidad una alternativa viable. Sin embargo hay todavía ciertos problemas, tales como la disipación de calor, stress térmico y ciertas consideraciones de diseño físicas, que todavía tienen que ser resueltos para dichas arquitecturas. La idea está basada en las técnicas de módulos multichip (actualmente utilizada en módulos de memoria de alta capacidad), donde se apilan un número de dispositivos FPGAs en 2D. El punto crítico serían los contactos realizados entre los distintos dispositivos que deberían ser realizados con vías que pasen a través de los mismos. Pero el número de conexiones verticales que pueden caber en un dispositivo determina el ancho y separación de los canales verticales que unen las capas de la FPGA.

Hay dos propuestas principales para definir las arquitecturas 3D:

- Apilando componentes. Primero, se crean distintos chips con procedimientos estándar, y posteriormente se unen de cara a cara, a través de la base o apilándolos para finalmente crear un sistema 3D. Dentro de esta metodología existen distintas opciones dependiendo del tipo de componente que se apila: dado sobre dado (*die on die*), dado

sobre oblea (*die on wafer*), etc. La alternativas de interconexión dependen del tipo de componente y pueden ser vias en obleas, conexiones metálicas en dados semiconductores (*metal bonding*) o con bolas de soldadura (solder pump). La figura A.2.6 muestra distintas opciones de apilamiento de dados y circuitos donde se utilizan conexiones metálicas, bolas de soldadura y vías.

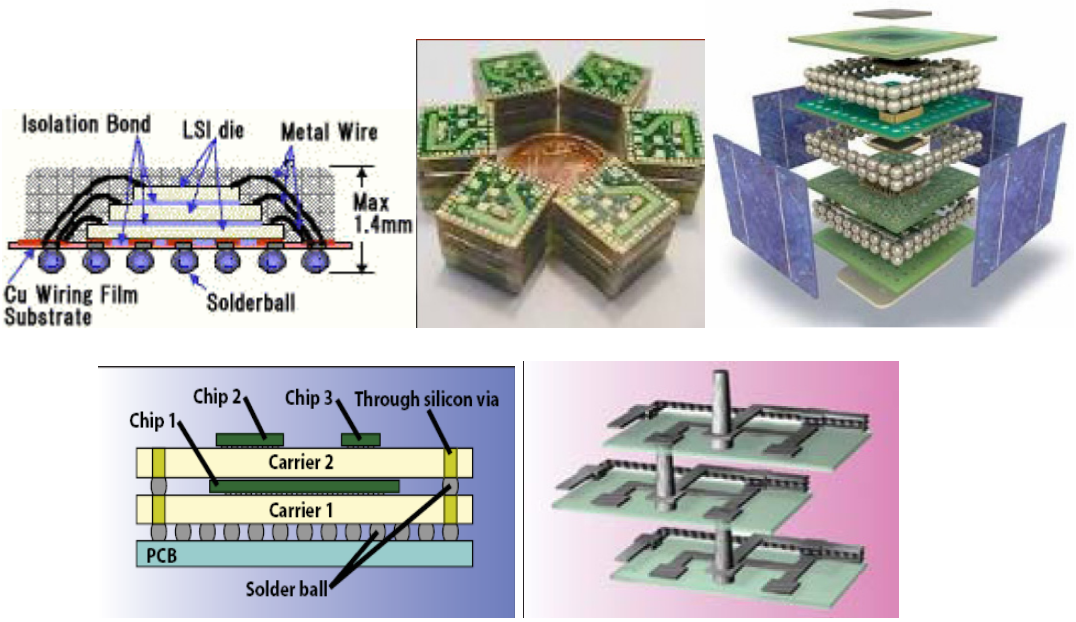


Figura A2.6. FPGA 3D con circuitos apilados.

- Empotrando todos los chips en una capa de pegamento que contiene el rutado e interconexiones entre los distintos módulos. Rahman [RDCR03] ha determinado que el retardo debido a la interconexión puede reducirse hasta un 60% y la disipación de potencia hasta un 55%.

El Consorcio Internacional de I&D Sematech propone la tecnología de interconexión 3D como un puente entre las interconexiones de cobre actuales y tecnologías futuras como nanotubos de carbón. Se propone realizar conexiones a través de vías en el *wafer*. Algunos fabricantes como Texas Instruments proponen separar la lógica de los dispositivos de memoria, y apilarlos para reducir la longitud de las conexiones, como se puede ver en la figura A2.7, y por tanto los retardos en el reloj aproximadamente entre un 30 y un 40%.

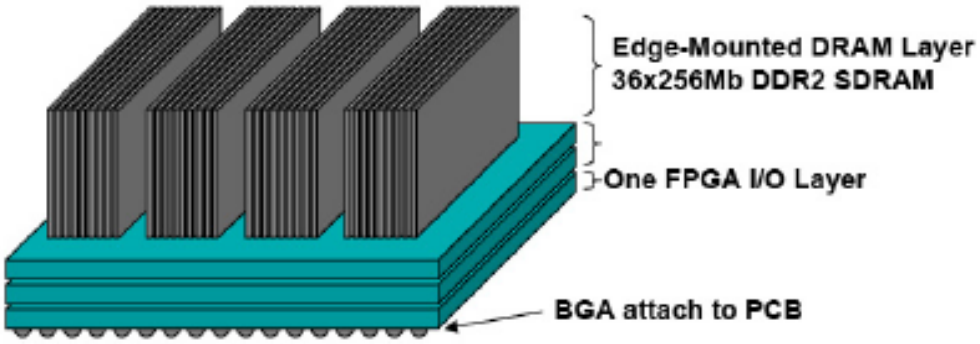


Figura A2.7. FPGA 3D.

En la actualidad se dispone de algunos prototipos de FPGAs 3D, como la fabricada por la Universidad de Cornell [FLPM06] en el que se apilan 3 FPGAs idénticas, o el prototipo desarrollado por Tezzaron [Tezz08] en el que se interconectan las distintas capas con una técnica denominada TSV (Through Silicon Vias) donde se interconectan las distintas capas a través de vías implementadas directamente en la oblea.

Por otra parte, hay una carencia de herramientas CAD para diseño de circuitos 3D que sean eficientes y que puedan explotar las ventajas potenciales que la integración 3D pueda ofrecer.

Actualmente se están desarrollando algunas herramientas CAD para explorar el potencial de estos sistemas, como la propuesta por Ababei en [AFGM05] donde se proponen algoritmos de colocación basados en *simulated annealing* y de partición orientado a minimizar retardos junto a una herramienta de rutado para FPGAs en 3D.

Para poder evaluar el efecto de las distintas posibilidades arquitectónicas, los diseñadores necesitan herramientas que puedan disponer de parámetros arquitectónicos como entradas, y poder reportar como salida longitudes de hilo, anchura de canal, área y retardo de los circuitos utilizados como bancos de prueba. Para ello han diseñado una herramienta para colocación y rutado denominada TPR (*Threedimensional Place and Route*).

La herramienta propuesta particiona primero un diseño utilizando un algoritmo que intenta minimizar la utilización de las conexiones verticales (de

tipo min-cut) [KAKS97], ya que no hay tantos recursos de conexión en vertical como en horizontal. Para realizar esta partición se divide el diseño en un número de subcircuitos balanceados/equilibrados igual al número de capas disponible en la integración en 3D.

La figura A2.8 muestra conceptualmente cómo se particiona y distribuye en niveles un circuito de los utilizados como banco de prueba. El paso inicial de "división en capas" se realiza usando el algoritmo min-cut hMetis. En el segundo paso se asignan las particiones a distintas capas con el objetivo no sólo de minimizar la longitud total de conexiones verticales sino el corte máximo entre dos capas consecutivas. Posteriormente se realiza la asignacion de recursos y rutado de manera independiente en cada capa

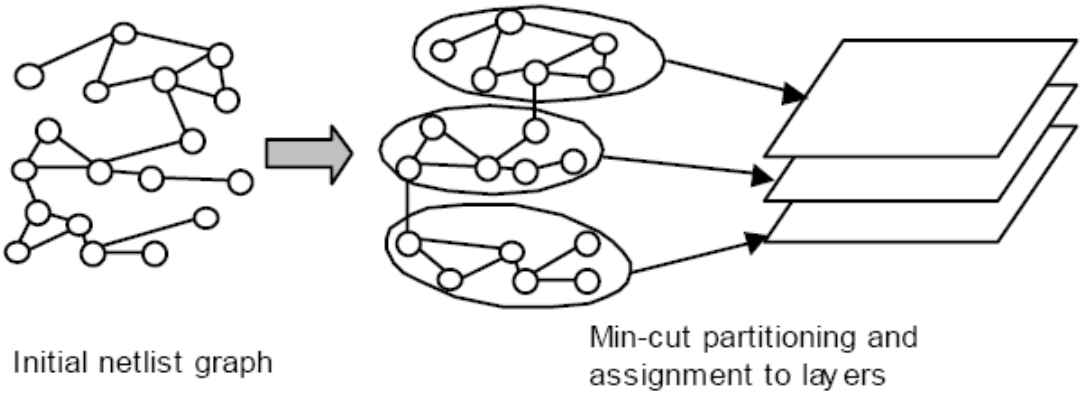


Figura A2.8. Partición de una netlist en diferentes niveles.

Después se utiliza la herramienta de colocación en cada capa, empezando con la capa superior y continuando hacia la capa inferior. El objetivo en este paso es minimizar ambos, la longitud total de conexión en vertical y el ancho máximo entre dos capas adyacentes. El algoritmo de rutado es una extensión del algoritmo VPR presentado en [BeRo97] para FPGAs en 2D. Los resultados muestran que la mejora en retardo obtenida para todos los circuitos del banco de pruebas, para un número de capas entre 5 y 6 obtiene grandes beneficios, pero para más capas la mejora es muy pequeña.

A2.4 FPGAs con memoria de configuración no-volátil

Otra posible mejora viene de las alternativas a las actuales memorias de reconfiguración, donde la mayoría de las FPGAs del mercado están basadas en SRAM, de tipo volátil, que tienen que ser cargadas con los datos de configuración desde memorias de arranque externas. Este proceso de descarga de los datos de configuración desde las memorias externas puede representar desde algunas decenas hasta cientos de milisegundos.

Las FPGAs basadas en memoria no-volátil almacenan los datos de configuración dentro del chip eliminando la necesidad de memoria externa y de los largos tiempos de configuración asociados. Una solución para proporcionar la no volatilidad ha sido la tecnología antifusible, pero aunque no son volátiles no pueden ser reconfiguradas: cada vez que su funcionalidad cambia se tiene que programar un nuevo dispositivo. Una alternativa es utilizar memoria flash como memoria de configuración como en los dispositivos de la familia LatticeXP que combinan memorias de configuración SRAM y Flash. La memoria SRAM controla la operación de la lógica y la Flash almacena los datos de configuración. En el arranque la memoria SRAM se carga desde la memoria Flash on-chip a través del bus de datos en menos de 1 ms. En la figura A2.9 se muestra la disposición de los dos tipos de memoria alrededor de la arquitectura y la ruta de datos masivamente paralela para reducir los tiempos de carga de la configuración.

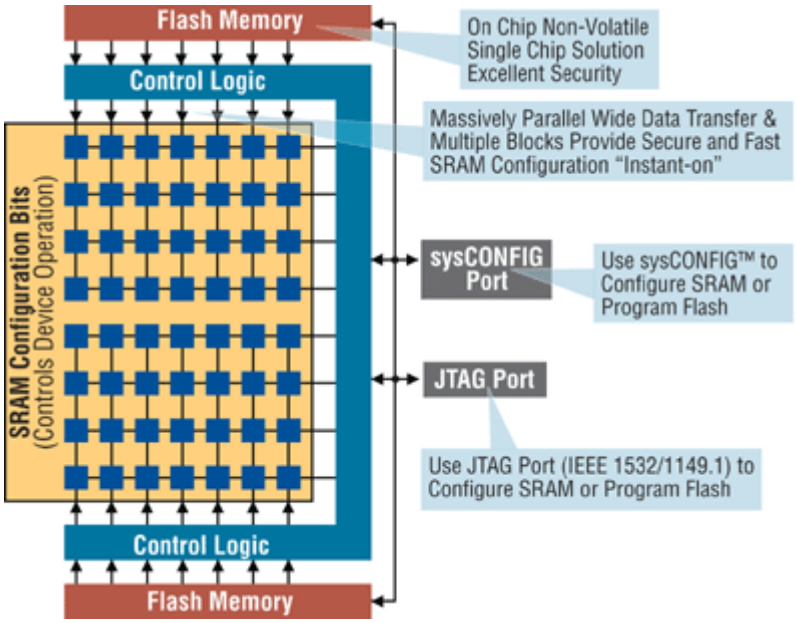


Figura A2.9. Memorias de configuración en la LatticeXP.

También se dispone de flash de configuración en las FPGAs de la familia Spartan-3 de Xilinx y ProAsic3 de Actel.